Axel Beckert <abe@cs.uni-sb.de>
Diplomarbeit: Kompilierung von Anytime-Algorithmen:
Konzeption, Implementation und Analyse
$Id: index.dvi,v 1.95 2000/12/20 07:19:15 abe Exp $

Axel Beckert <abe@cs.uni-sb.de>
Fachrichtung 6.2 Informatik
Universität des Saarlandes
Postfach 151 150
66041 Saarbrücken, Deutschland
http://w5.cs.uni-sb.de/~abe/

Diplomarbeit

Kompilierung von Anytime-Algorithmen: Konzeption, Implementation und Analyse

Axel Beckert

20. Dezember 2000

Lehrstuhl für Künstliche Intelligenz, Prof. Dr. Dr. h. c. Wolfgang Wahlster
Fachrichtung 6.2 Informatik, Naturwissenschaftlich-Technische Fakultät I
Universität des Saarlandes

Dieses Dokument wurde auf azka.deuxchevaux.org unter Linux 2.0 mit LaTeX 2_ε unter Verwendung der KOMA-Skript-Klasse scrbook als Dokumentenklasse und den Paketen a4, algorithm, algorithmic, apacite (modifiziert), babel, colordvi, dsfont, fancyhdr, lgrind, makeidx und theorem sowie BibTeX, MakeIndex, CVS, LGrind (modifiziert), bbfig (modifiziert), PERL 5 und GNU Make erstellt. Als Editor wurde ausschließlich GNU Emacs mit AUC TeX verwendet. Die Abbildungen in dieser Arbeit wurden mit Orcan V2 generiert, manuell in PostScript® programmiert oder mit xfig 3.2 auf azu.deuxchevaux.org unter Linux 2.0 erstellt und nach PostScript® exportiert.

Hast Du Deine Arbeit in TEX oder in einem elektronischen Format geschrieben?

(Antonio Krüger, 28. Januar 1999)

Eidesstattliche Erklärung

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Saarbrücken, den 20. Dezember 2000

Axel Beckert

Inhaltsverzeichnis

1. Einführung

In nahezu allen Bereichen des täglichen Lebens sind die uns zur Verfügung stehenden Ressourcen — z. B. Zeit, Platz, Arbeitskräfte, Geld, Geschwindigkeit, etc. — beschränkt. Eine Firma muß bei der Auswahl ihrer Zulieferer nicht nur beachten, wie gut deren Produkte sind, sondern auch, wie weit deren Produktionsstätte von dem Ort der Weiterverarbeitung entfernt ist und wieviel Zeit und Geld der Transport zwischen den beiden Orten kostet. In der Kryptographie basieren sehr viele Verschlüsselungsverfahren darauf, daß potentielle Gegner nicht genügend Ressourcen zum Austesten aller theoretisch möglichen Schlüssel haben.

Entsprechend muß mit den vorhandenen Ressourcen bewußt umgegangen und wenn möglich im voraus überlegt werden, wie sie auf die zu erledigenden Aufgaben verteilt werden können, um ein möglichst gutes Endergebnis zu bekommen.

Dazu reichen nicht immer, aber häufig, auch approximative Ergebnisse. Um die Lösung eines Problems anzunähern, werden oft wesentlich weniger Ressourcen gebraucht, als für die exakte Berechnung der Lösung notwendig wären. Besteht der Lösungsprozeß aus mehreren zu lösenden bzw. approximierenden Teilaufgaben, so sollte abgewogen werden, wo Ressourcen eingespart werden können, ohne große Abstriche beim Ergebnis machen zu müssen oder — andersherum gesagt — bei welchen Teilaufgaben sich der Einsatz von vielen Ressourcen am ehesten lohnt.

1.1. Einordnung

Diese Arbeit ist entstanden als Diplomarbeit im Rahmen des von der Deutschen Forschungsgemeinschaft (DFG) geförderten Sonderforschungsbereich 378, *„Ressourcenadaptive kognitive Prozesse“* [SFB, 1997] an der Naturwissenschaftlich-Technischen Fakultät I der Universität des Saarlandes in Saarbrücken. Dort wird im Teilprojekt A4 „Ressourcenadaptive kognitive Prozesse“ REAL[1] am Beispiel der Erzeu-

[1] Das Akronym REAL steht für „Ressourcenadaptive Lokalisation“.

gung sprachlicher Raumbeschreibungen in Dialogsituationen das Interagieren von ressourcenbeschränkter Objektlokalisierung und inkrementeller Sprachproduktion untersucht [REAL, 1996; Wahlster, Blocher, Baus, Stopp & Speiser, 1998].

Das in diesem Kontext entwickelte System BOLA[2] [Blocher, 1999] soll auf Fragen des Benutzers nach der Lage eines Objektes im Raum diese anhand von Referenzobjekten beschreiben können. Solche Systeme sind geeignet als Auskunftsysteme auf Flughäfen, Messen oder großen Bahnhöfen, aber auch für die Verwendung in Navigationshilfen von Fahrzeugen [Maaß, 1996] oder zur Kommunikation mit autonomen Robotern [Stopp, 1998]. Da — je nach Situation – mal schnelle (z. B. falls der Flug oder Zug des Benutzers in Kürze geht) und mal detaillierte Antworten (z. B. Beschreibung eines Weges einmal quer durch das Flughafenterminal für eine Person, die zum ersten Mal dort abfliegt oder landet) gefordert sind, sollte sich das System an zeitliche Anforderungen des aktuellen Benutzers anpassen können. Hierfür hat sich eine sogenannte Anytime-Architektur angeboten, welche es ermöglicht, die Berechnung der Antwort jederzeit zu unterbrechen und eine bis dahin erreichte, gegebenenfalls weniger detaillierte Antwort auszugeben, ohne das Endergebnis abwarten zu müssen.

In dieser Arbeit wird ein System zur Ressourcenverteilung auf Anytime-Algorithmen vorgestellt, welches durch die Approxmierung der Verteilung versucht, mit einem geringen eigenen Ressourcenverbrauch möglichst nahe an die optimale Ressourcenverteilung heranzukommen.

1.2. Anforderungen und Ziele

Die Verwendung von Anytime-Algorithmen kann in unterschiedlichsten, komplexen Systemen erfolgen. Dabei sind die Verfahren zur Optimierung der Ressourcenverteilung unabhängig von der konkreten Problemstellung. Daher ist es angebracht, diese Optimierung als eigenständiges (Sub-) System aufzufassen, das über definierte Schnittstellen mit den verschiedenen problemlösenden Hauptsystemen kommuniziert.

Kern des in dieser Arbeit beschriebenen Systems ORCAN[3] V2 ist eine Common-Lisp-Funktion, über welche ein übergeordnetes System (wie z. B. „JAMES“[4] [Wittig, 1998]) die Ressourcenverteilung für Anytime-Algorithmen kompilieren[5] lassen kann.

[2] Beschränkt-optimaler Lokalisationsagent
[3] Operational Rationality through Compilation of Anytime Algorithms
[4] Java Anytime Management & Editor System
[5] Um die Kompilierung von Anytime-Algorithmen nach [Zilberstein, 1993] von der Compilie-

Dazu wird dieser Funktion Meta-Wissen über die Anytime-Algorithmen in Form von Performanzprofilen und gegebenenfalls auch Verknüpfungsfunktionen als Parameter übergeben. ORCAN V2 versteht mehrere Repräsentationsformen von Performanzprofilen und bietet zudem verschiedene Kompilierungsmethoden an.

ORCAN V2 baut auf dem in [Baus & Beckert, 1998] beschriebenen System ORCAN (V1) auf und umfaßt u. a. weitere mögliche Eingabeformate für Performanzprofile, eine neue Methode zur Kompilierung von Anytime-Algorithmen, die Möglichkeit, beliebige Verknüpfungsfunktionen zu verwenden, eine spezielle Kompilierung bei besonders wenig zur Verfügung stehenden Ressourcen sowie viele Detailverbesserungen.

Zum Vergleichen und Austesten von Kompilierungsmethoden bietet ORCAN V2 außerdem eine Testumgebung an, welche zufällige Performanzprofile generieren, Laufzeitmessungen[6] der Kompilierungsmethoden durchführen sowie die Ergebnisse der Verteilungen und Laufzeitmessungen grafisch darstellen kann.

1.3. Gliederung der Arbeit

Im anschließenden Kapitel werden die Grundlagen von Ressourcenbeschränkung und Anytime-Algorithmen erklärt sowie für diese Arbeit wichtige Forschungsergebnisse aufgeführt. Daraus ausgegliedert sind die Kompilierungsmethoden. Sie werden in Kapitel 3 näher vorgestellt.

Kapitel 4 stellt die Anforderungen an ORCAN V2 sowie dessen Konzeption vor. Darin enthalten sind auch neue Eingabeformate bzw. Darstellungsformen von Performanzprofilen sowie neue Kompilierungsmethoden. In Kapitel 5 wird die Implementierung von ORCAN V2 beschrieben.

In Kapitel 6 findet sich die Analyse der Kompilierungsmethoden unter Verwendung der mit ORCAN gemachten Laufzeittests sowie Betrachtungen bezüglich der Verwendbarkeit dieser Ergebnisse in ressourcenadaptierenden Systemen.

Abschließend faßt Kapitel 7 die Ergebnisse dieser Arbeit zusammen und zeigt Ansätze für weiterführende Arbeiten auf diesem Gebiet.

In Anhang A finden sich weitere, nicht ausführlicher erläuterte Laufzeittests, die

rung von Programmen im üblichen Sinne (also der Übersetzung von Quelltext in Byte- oder Maschinencode) zu unterscheiden, wird in dieser Arbeit für erstere die Schreibweise mit „K“ verwendet.

[6] Für Laufzeitmessungen ist Lucid Common Lisp oder Liquid Common Lisp notwendig.

zur Untermauerung der in Kapitel 5 und 6 gezogenen Schlüsse dienen. Anhang B enthält die Beschreibung der Parameter der von ORCAN dem Benutzer bzw. übergeordneten System zur Verfügung gestellten Funktionen. Anhang C enthält Teile des Common-Lisp-Quelltextes der Kompilierungsmethoden. Abschließend sind in Anhang D Abbildungs-, Tabellen-, Algorithmen- und Stichwortverzeichnis sowie die Literaturangaben zu finden.

2. Ressourcenbeschränkung und Anytime-Algorithmen

Bis vor nicht allzulanger Zeit wurden in der Künstlichen Intelligenz die für eine Berechnung notwendigen Ressourcen als nahezu unbeschränkt verfügbar betrachtet. Seit einigen Jahren berücksichtigen Forschungsansätze nun auch mögliche Einschränkungen bezüglich der verwendbaren Ressourcen, z. B. bei Echtzeitsystemen wie autonomen Robotern [Stopp, 1998] oder Navigationshilfen in Fahrzeugen [Maaß, 1996].

2.1. Ressourcenbegriff

Unter Ressourcen versteht man nach [Jameson & Buchholz, 1998] im kognitionswissenschaftlichen Sinne Hilfsmittel, die verwendet werden, um ein bestimmtes Ziel zu erreichen. Ressourcen können dabei unterschiedliche Eigenschaften besitzen: Es gibt z. B. Ressourcen, die verbraucht werden (z. B. Zeit, Strom, etc.) und Ressourcen, die im Normfall beliebig oft verwendet werden können (z. B. Bandbreite, Speicherplatz, Inhalte von Datenbanken, aber auch menschliche Eigenschaften wie Lesen, Schreiben, etc.); genauso gibt es Ressourcen, die (nahezu) beliebig klein unterteilt werden können (z. B. Zeit im allgemeinen Sinn) und Ressourcen, bei denen eine Unterteilung nur bis zu einer bestimmten Grenze möglich ist (z. B. einzelnen Rechenschritten).

Aus diesem Ressourcenbegriff ergeben sich für Jameson und Buchholz (1998) bezüglich des Forschungsgebietes des Sonderforschungsbereiches 378, *„Ressourcenadaptive kognitive Prozesse“*, u. a. folgende für diese Arbeit wichtige Fragen:

- Wie können die zur Verfügung stehenden Ressourcen effektiv auf die zu erledigenden Aufgaben verteilt werden?
- Wie können zur Verfügung stehende Ressourcen effektiv auf zu erledigende

Aufgaben verteilt werden, auch wenn die Menge der zur Verfügung stehenden Ressourcen im voraus nicht bekannt ist?

Die Kompilierung von Anytime-Algorithmen ist eine mögliche Antwort auf diese beiden Fragen. Dies wird in den folgenden Abschnitten näher erläutert.

2.2. Ressourcensensitivität

Von Ressourcensensitivität spricht man, wenn zur Lösung eines Problems die Beschränkung der zur Verfügung stehenden Ressourcen in beliebiger Form berücksichtigt wird [Wahlster & Tack, 1997]. Dabei werden drei Arten der Berücksichtigung von Ressourcenbeschränkung unterschieden:

Ressourcenadaptiertes Verhalten: Das Verhalten ist auf eine feste und im voraus bekannte Ressourcenbeschränkung optimiert und kann sich nicht an Beschränkungen anpassen, die sich während des Ablaufs ändern. Dies bedeutet einerseits, daß sich die Ergebnisqualität bei gleichbleibender Eingabequalität nicht verbessert, auch wenn mehr Ressourcen zur Verfügung stehen, als die Ressourcenmenge, auf die das Verhalten hin optimiert wurde. Andererseits bedeutet dies auch, daß das gegebene Problem gar nicht gelöst wird, falls die zur Verfügung stehende Ressourcenmenge einmal unter die für die Optimierung angenommene Menge sinkt, da der Vorgang innerhalb der Ressourcenbeschränkung nicht beendet werden kann.

Ressourcenadaptives Verhalten: Es gibt eine im voraus festgelegte Verhaltensstrategie, mit der die Verteilung von Ressourcen auf die verschiedenen Arbeitsschritte anhand von Meta-Wissen und der aktuellen Ressourcenbeschränkung geschieht. Der Unterschied zum ersten Fall liegt darin, daß die Ressourcenbeschränkung variieren kann und dies berücksichtigt wird, verschiedene Ressourcenbeschränkungen also auch verschiedene Ressourcenverteilungen zur Folge haben können.

Ressourcenadaptierendes Verhalten: Wie im zweiten Fall werden durch das Vorhandensein von Meta-Wissen sowie durch Informationen über die aktuellen Ressourcenbeschränkungen Entscheidungen über die Verteilung von Ressourcen dynamisch getroffen. Im Unterschied zum vorherigen Fall werden aber auch Entscheidungen bezüglich der zu verwendenden Strategien bzw. Algorithmen dynamisch gefällt. Dies ermöglicht es z. B., bei geringen zur Verfügung stehenden Zeitressourcen eine ungenauere, aber schnellere Berechnungsmethode zu verwenden, als in einem Fall, in dem mehr Zeit zur Verfügung steht.

Eine grafische Darstellung der Klassifizierung der genannten Verhaltensweisen findet sich in Abbildung 2.1.

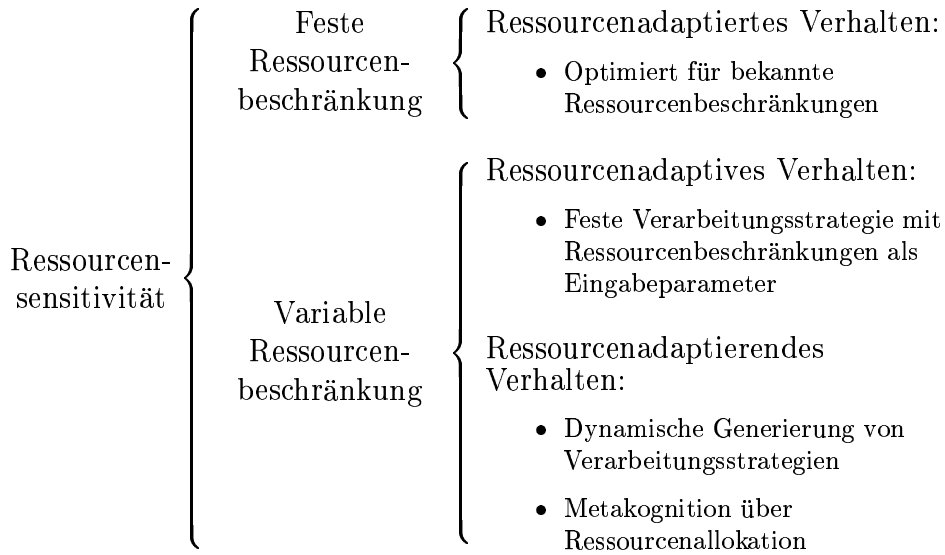

Abbildung 2.1.: Klassifizierung der Ressourcensensitivität nach [Wahlster, 2000]

2.3. Anytime-Algorithmen

Um vor allem zeitlichen, aber auch anderen Ressourcenbeschränkungen gerecht zu werden, entstanden verschiedene Ansätze, ressourcensensitive Berechnungstechniken zu realisieren.

Auf den *flexiblen Berechnungen* von Horvitz (1987) aufbauend, entwickelten Dean und Boddy (1988) das Konzept der Anytime-Algorithmen. Danach werden Anytime-Algorithmen durch die drei folgenden Eigenschaften charakterisiert:

i. Ihre Ausführung läßt sich durch Scheduling-Techniken steuern, d. h. ihre Ausführung läßt sich mit geringem Aufwand von außen gesteuert aussetzen und wieder aufnehmen.

ii. Ihre Ausführung läßt sich jederzeit beenden, und sie geben dabei auf jeden Fall ein Ergebnis zurück.

iii. Die Qualität[1] der zurückgegebenen Antworten steigt monoton mit der investierten Rechenzeit.

[1] Der Begriff Qualität wird in Abschnitt 2.3.2 näher erläutert.

Entsprechend läßt sich ein *Anytime-Algorithmus* wie folgt definieren:

Definition 2.1 (Anytime-Algorithmus): *Ein* Anytime-Algorithmus *ist ein Algorithmus, der jederzeit unterbrochen werden kann, wobei die Qualität des dabei zurückgegebenen (Zwischen-) Ergebnisses mit der investierten Rechenzeit monoton steigt.*

Vor allem die letzten beiden der drei Eigenschaften machen den Unterschied gegenüber normalen Algorithmen aus. Das heißt aber nicht, daß Anytime-Algorithmen kompliziert zu entwickeln sind. Im Gegenteil: Es gibt bereits viele bekannte — meist iterative — Algorithmen, die sich mit geringen Modifikationen in Anytime-Algorithmen umwandeln lassen. Ein beliebtes Beispiel hierfür sind rekursive Näherungsverfahren, wie z. B. das Newtonsche Verfahren zur Nullstellen-Bestimmung und sogenannte Hillclimbing-Algorithmen, da hier eine Unterbrechung problemlos möglich ist, beispielsweise als Abbruchbedingung der Rekursion.

Aufgrund der obengenannten Eigenschaften eignen sich Anytime-Algorithmen besonders gut für Systeme mit variabler Ressourcenbeschränkung wie z. B. ressourcenadaptive und -adaptierende. Für die Nutzung von Anytime-Algorithmen in solchen Systemen muß allerdings noch das Meta-Wissen über die jeweils verwendeten (bzw. verwendbaren bei ressourcenadaptierenden Systemen) Anytime-Algorithmen zur Verfügung gestellt werden. Hierzu werden sogenannte Performanzprofile genutzt, die in Abschnitt 2.3.1 ausführlich erläutert werden.

Zunächst wird aber noch eine Variante der Anytime-Algorithmen, die Contract-Algorithmen, beschrieben. Auf den Anytime-Algorithmen nach Dean und Boddy (1988) baut Zilberstein (1993) das Konzept der *Contract-Algorithmen* auf. Sie unterscheiden sich von *echten Anytime-Algorithmen*[2] durch die Einschränkung, daß ihnen zu Beginn ihrer Berechnung, z. B. als Parameter, mitgeteilt wird, wieviel Zeit ihnen für ihre Berechnung zur Verfügung steht. Vor Ablauf dieser Zeitspanne müssen sie keine Ergebnisse zurückliefern. Ihnen wird sozusagen „vertraglich“ (daher die Bezeichnung „Contract“) zugesichert, daß sie eine bestimmte Zeitspanne für ihre Berechnung verwenden können.

Grund für diese zusätzliche Einschränkung ist, daß es bei komplexen Anforderungen oft einfacher ist, einen Contract-Algorithmus anstatt eines echten Anytime-Algorithmus zu implementieren. Allerdings möchte man im Normalfall nicht auf die Unterbrechbarkeit der echten Anytime-Algorithmen verzichten. Die Lösung dieses

[2] Zilberstein (1993) teilt die Anytime-Algorithmen in zwei Gruppen auf: *Contract-Algorithmus* und *unterbrechbare Algorithmen*. Letztere entsprechend den Anytime-Algorithmen nach der Definition in [Dean & Boddy, 1988]. Um Mißverständnisse zu vermeiden, werden in dieser Arbeit dort, wo es zur Unterscheidung hilfreich ist, die Anytime-Algorithmen nach der Definition von Dean und Boddy (1988) als *echte Anytime-Algorithmen* bezeichnet.

Problems liefert der *Reduktionssatz von Zilberstein*, welcher zeigt, daß jeder Contract-Algorithmus in einen echten Anytime-Algorithmus umgewandelt kann:

Satz 2.1 (Reduktionssatz von Zilberstein): *Für jeden Contract-Algorithmus CA mit Qualität $\mathrm{q}_{CA}(t)$ kann ein echter Anytime-Algorithmus AA mit Qualität $\mathrm{q}_{AA}(t)$ konstruiert werden, so daß für alle t gilt:*

$$\mathrm{q}_{AA}(4t) = \mathrm{q}_{CA}(t) \tag{2.1}$$

Der Satz basiert darauf, daß ein echter Anytime-Algorithmus AA gebaut wird, welcher den Contract-Algorithmus CA hintereinander mit exponentiell steigenden Laufzeitwerten als Parameter aufruft und dessen errechnetes Ergebnis bis zum nächsten zwischenspeichert sowie gegebenenfalls zurückgibt. Ein ausführlicher Beweis kann in [Zilberstein, 1993, Seite 38] gefunden werden.

2.3.1. Performanzprofile

Wie in Abschnitt 2.3 erwähnt, muß ein ressourcenadaptives System Meta-Wissen über die verwendeten Anytime-Algorithmen haben. Hierfür eignet sich gut die zu erwartende Qualität eines Ergebnisses nach einer bestimmten Laufzeit. Die Qualität eines Zwischenergebnisses kann in Prozent relativ zum optimalen Ergebnis bzw. als Zahl im auf das optimale Ergebnis normierten Intervall $[0;1]$ ausgedrückt werden; der Qualitätsverlauf eines Anytime-Algorithmus läßt sich dann in Form von *Performanzprofile* genannten Funktionen $\mathrm{q}_{AA} : \mathbb{R}_0^+ \longmapsto [0;1]$ angeben. Da nach Definition 2.1 die Qualität eines Anytime-Algorithmus monoton steigend ist, ergibt sich für Performanzprofile folgende Definition:

Definition 2.2 ***Performanzprofil:*** *Ein* Performanzprofil *beschreibt die Qualitätsentwicklung eines Anytime-Algorithmus AA im Verhältnis zur zu Berechnung nutzbarer Zeit in Form einer monoton steigenden Funktion* $\mathrm{q}_{AA} : \mathbb{R}_0^+ \longrightarrow [0;1]$

Folgende Definitionen werden ebenfalls im weiteren verwendet:

Definition 2.3

$$\mathbb{P} := \textit{Menge aller zur Kompilierung verwendeten Performanzprofile}$$

Definition 2.4

$$\mathbb{V} := \textit{Menge der möglichen Ressourcenverteilungen auf } \mathbb{P}$$

Technischer Hinweis: Zur grafischen Darstellung und Veranschaulichung der Performanzprofile werden Diagramme mit Qualitäts- und Ressourcen-Achse verwendet — falls die Ressource Zeit ist, auch mit Zeit- anstatt Ressourcen-Achse. Aus diesem Grunde werden in dieser Arbeit anstatt der üblichen x- und y-Achsen r- bzw. t- und q-Achsen verwendet, wobei die r- bzw. t- der x- und die q- der y-Achse entspricht.

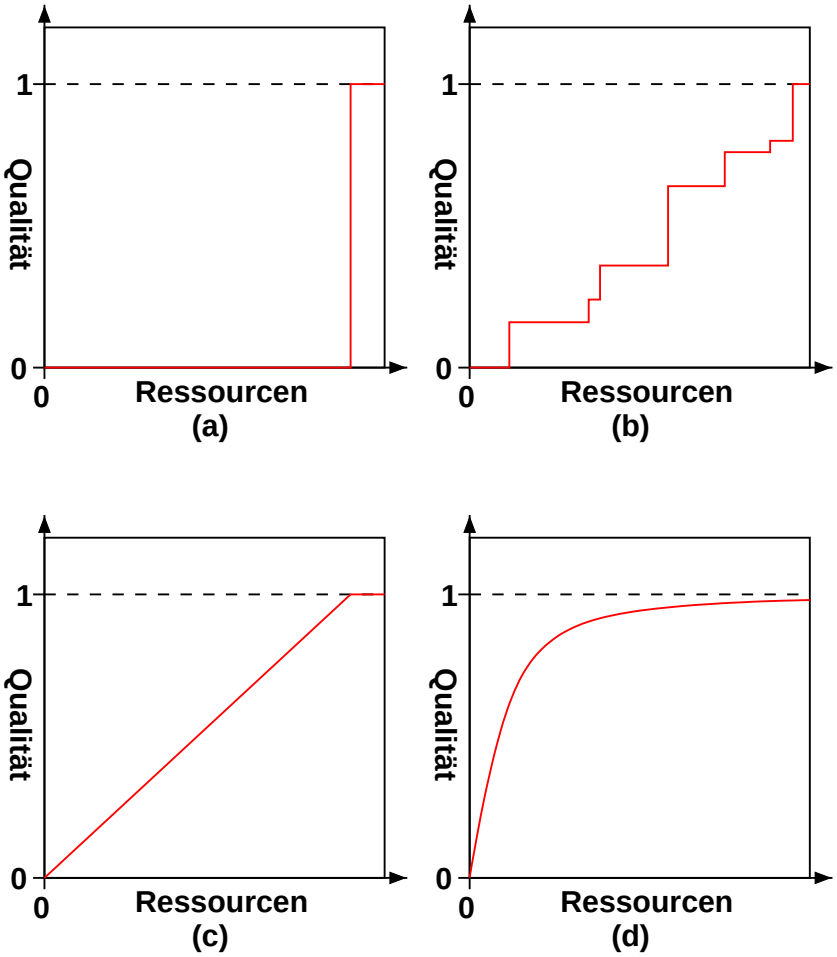

Abbildung 2.2.: Beispiele für Performanzprofile

Abbildung 2.2 zeigt einige einfache Beispiele für Performanzprofile:

Beispiel (a) zeigt das Performanzprofil eines typischen Standard-Algorithmus: Nach einem bestimmten Zeitpunkt ist das Endergebnis errechnet, die Qualität steigt sprunghaft von 0% auf 100%.

Beispiel (b) zeigt das Performanzprofil der Implementation eines typischen Anytime-Algorithmus: Die Qualität wird durch Verfeinerung Stück für Stück ge-

steigert, bis sie 100% erreicht hat. Das Performanzprofil erinnert an eine Treppe, auch wenn es aus unregelmäßig großen Treppenstufen bestehen kann.

Beispiel (c) zeigt das Performanzprofil eines idealisierten Anytime-Algorithmus, welcher seine Qualität kontinuierlich steigert, bis die Qualität von 100% erreicht wird. Solche Performanzprofile lassen sich z. B. durch eine Funktion der Art $\mathrm{q}(r) = \min(1; \max(0; mr + q_0))$ darstellen.

Beispiel (d) zeigt das Performanzprofil eines Näherungsverfahrens auf Basis eines Anytime-Algorithmus: Die Qualität steigt kontinuierlich, erreicht jedoch im Normalfall nie 100%. Solche Performanzprofile lassen sich häufig durch eine Funktion der Art $\mathrm{q}(r) = 1 - e^{-\lambda r}$ darstellen.

Wie in Abbildung 2.2b gezeigt, steigern Anytime-Algorithmen ihre Qualität nie kontinuierlich, sondern immer schrittweise: Solange ein Anytime-Algorithmus dabei ist, sein aktuelles Zwischenergebnis zu verbessern, wird dieses zwischengespeichert und gegebenenfalls auf Anfrage zurückgegeben. Entsprechend sieht der Graph eines Performanzprofils — gegebenenfalls nach entsprechender Vergrößerung — treppenstufenförmig aus.

Folgen diese Qualitätsverbesserungen jedoch im Gegensatz zu dem Beispiel in Abbildung 2.2b sehr kurz aufeinander, so kann die Qualitätsentwicklung ohne große Fehler durch eine stetige Funktion angenähert werden, wie sie z. B. in den Abbildungen 2.2c und d — gegebenenfalls mit einer Beschränkung nach oben und unten wie in Beispiel (c) — gezeigt werden. Entsprechend werden stetige Funktionen auch als Darstellungsformen für Performanzprofile verwendet. Diese und andere Darstellungsformen werden in Abschnitt 2.5 näher behandelt.

2.3.2. Qualitätsmetriken

Es mag beim Lesen des letzten Abschnitts die Frage aufgekommen sein, was denn unter Qualität genau zu verstehen ist. Im Falle von Anytime-Algorithmen ist Qualität eine allgemeine Bewertung für die Güte eines Erzeugnisses oder Ergebnisses. Sie kann — je nachdem um was für ein System es sich handelt — unterschiedlichste Metriken annehmen.

In bezug auf Anytime-Algorithmen haben Russell und Zilberstein (1996) die Qualitätsmetriken in drei Gruppen aufgeteilt:

Genauigkeit: Die Qualitätsmetrik *Genauigkeit* gibt an, wie nahe das errechnete Ergebnis am exakten Ergebnis liegt, und bietet sich vor allem für numerische

Werte an. Beispiel für einen Maßstab dieser Art von Qualität wären z. B. die Anzahl der Nachkommastellen des Ergebnisses.

Sicherheit: Die Qualitätsmetrik *Sicherheit* entspricht der Wahrscheinlichkeit, daß das errechnete Ergebnis (z. B. eine Diagnose bei einer medizinischen Untersuchung) korrekt ist. Sie ist vor allem für Ergebnisse mit Booleschen Werten oder Werten aus einer endlichen Menge geeignet.

Spezifität: Die Qualitätsmetrik *Spezifität* beschreibt, wie detailliert bzw. komplex das Ergebnis ist. Maßstab hierfür kann beispielsweise die Tiefe eines Baumes sein, der als Ergebnis erstellt werden soll. Ein Anwendungsbeispiel für diese Metrik wäre eine Erläuterung, die den Radwechsel an einem Auto beschreibt: Sie kann sehr knapp ausfallen oder bis in das kleinste Detail gehen.

Wahlster (2000)[3] erweitert diese Liste noch um zwei Punkte:

Allgemeinheit: Die Qualitätsmetrik *Allgemeinheit* gibt an, wie allgemeingültig z. B. eine Problemlösung ist. Beispiel hierfür wäre ein mathematischer Beweis: Er kann für einen einzigen Spezialfall gelten oder einen sehr großen Gültigkeitsbereich haben.

Vertrauen: Die Qualitätsmetrik *Vertrauen* zeigt, wie hoch das Vertrauen in das Ergebnis ist. Klassisches Beispiel sind hierfür experimentelle Versuchsreihen, bei denen das Vertrauen in die Ergebnisse mit der Anzahl der Versuche wächst.

2.3.3. Transaktionen

Eine der Anforderungen an einen Anytime-Algorithmus ist, jederzeit unterbrechbar zu sein. Nun ist dies aber bei vielen Abläufen nicht uneingeschränkt möglich, da bestimmte Berechnungen nicht unterbrochen werden dürfen, weil sonst unerwünschte Nebeneffekte auftreten können[4]. Diese Beschränkung wird durch das Transaktionskonzept von Görz und Kesseler (1994) formalisiert. Danach ist eine *Transaktion* wie folgt definiert:

Definition 2.5 (Transaktion): *Eine* Transaktion *ist eine Folge von Berechnungsschritten, die nicht unterbrochen werden kann.*

[3] Siehe auch [Blocher, 1999, Seite 64].

[4] Wittig (1998) nennt als Beispiel für solch eine Transaktion das Überweisen von Geld von einem auf ein anderes Konto. Wird der Ablauf vorzeitig unterbrochen, so ist es möglich, daß das Geld vom ersten Konto bereits abgezogen wurde, aber auf dem zweiten Konto noch nicht hinzuaddiert und damit die Gesamtmenge an Geld nicht mehr konstant ist.

Soll ein aus Transaktionen bestehender Algorithmus gebaut werden, so bieten sich als Subkomponenten dafür die in Abschnitt 2.3 beschriebenen Contract-Algorithmen an, da ihnen bereits zu Beginn bekannt gemacht werden muß, wieviel Ressourcen ihnen zur Verfügung stehen und sie damit berechnen können, welche Transaktionen noch innerhalb der gegebenen Ressourcen zu Ende geführt werden können.

Sofern die Nebenwirkungen der Transaktionen nicht zeitkritisch sind, kann die Beendigung angefangener Transaktionen auch ausgeführt werden, nachdem der zeitkritische Teil des Systems fertig ist.

Eine weitere Möglichkeit, Anytime-Algorithmen mit Transaktionen so zu behandeln, daß die Transaktionen nicht unterbrochen werden, wird im Abschnitt 4.3.2.2 aufgeführt.

2.4. Erzeugung von Performanzprofilen

Performanzprofile von Anytime-Algorithmen können auf unterschiedlichste Weise erzeugt werden[5].

Die wohl am häufigsten genutzte Möglichkeit ist die Erzeugung von Performanzprofilen durch die statistische Auswertunge von Laufzeittests. Dabei werden statistische Daten über die zeitliche Entwicklung der Qualität gesammelt. Je nach Qualitätsmetrik wird z. B. über Mittel- oder Minimum-, aber auch über Maximumfunktionen ein eindeutiges Performanzprofil berechnet. Diese Variante kann auch eingesetzt werden, um Performanzprofile während der Anwendung weiter zu präzisieren [Zilberstein, 1993].

Eine andere Möglichkeit, die sich bei vielen iterativen Algorithmen anbietet, sind strukturelle Analysen des Algorithmus. Ist zum Beispiel „Genauigkeit“ die Qualitätsmetrik, so kann beim Newtonschen Näherungsverfahren der Fehler anhand der Anzahl der Iterationen abgeschätzt werden [Zilberstein, 1993].

Eine weitere Möglichkeit sind Projektionen. Diese können eingesetzt werden, wenn Teile des Performanzprofils bereits vorhanden sind. Beispielsweise kann am letzten bekannten Punkt eine Tangente mit der zuletzt bekannten Steigung angelegt und das Performanzprofil so verlängert werden [Wittig, 1998].

[5] vgl. [Grass, 1996]

2.5. Repräsentationsformen von Performanzprofilen

Performanzprofile von Anytime-Algorithmen können in unterschiedlichen Formen dargestellt bzw. gespeichert werden. Dean und Boddy (1988) geben als Beispiele für Performanzprofile sowohl Sprungfunktionen als auch exponentielle Funktionen der Art $\mathrm{q}(r) = 1 - e^{-\lambda x}$ und lineare Funktionen an. Diese Funktionen können z. B. jeweils durch ihre Funktionsklasse und deren Parameter dargestellt werden.

Im folgenden werden verschiedene Repräsentationsformen von Performanzprofilen beschrieben.

2.5.1. Stützpunkte

Als *Stützpunkte* werden Punkte bezeichnet, die ein geometrisches Element — hier den zweidimensionalen Graph eines Performanzprofils — charakterisieren. Abbildung 2.3a zeigt die Stützpunkte des in Abbildung 2.3b gezeigten Performanzprofils. In der Praxis wurde bisher fast immer ein Graph verwendet, der zwischen den Stützpunkten linear verläuft. Dies ist zwar naheliegend, aber nicht notwendig.

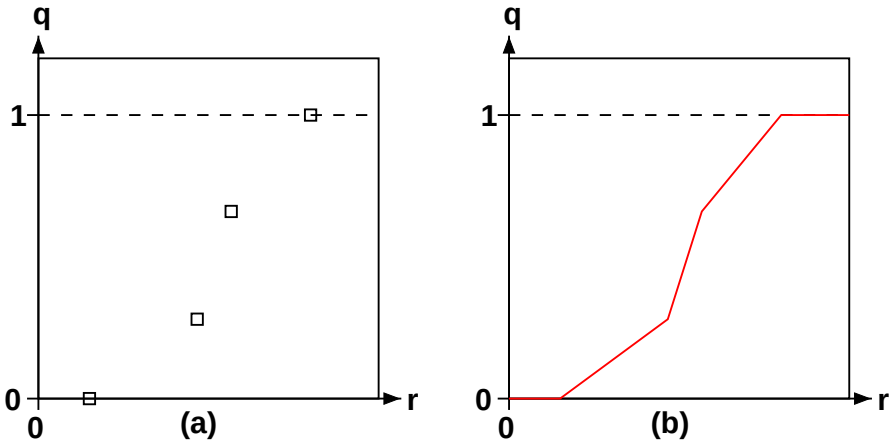

Abbildung 2.3.: Stützpunkte und ein daraus resultierende Performanzprofil

Wird ein Performanzprofil, wie in Abschnitt 2.4 beschrieben, aus statistischen Daten erzeugt, so bieten sich als Darstellungform Stützpunktlisten an. Dabei entsprechen die beiden Koordinaten der Stützpunkte der Qualität q und der Ressourcenmenge r, bei der die Qualität q erreicht wird. Die Reihenfolge der Stützpunkte ist durch ihre r-Werte festgelegt.

Diese Darstellungsart der Performanzprofile entspricht genau dem stützstellenartigen Wissen, das sie darstellt, denn zwischen zwei aufeinanderfolgenden, gemessenen Qualitätswerten kann der Verlauf nur in wenigen Fällen[6] genau bestimmt werden. In vielen Fällen kann für den Qualitätswert aufgrund der Monotonie nur ein Intervall angeben werden, innerhalb dessen der Qualitätswert liegen muß.

Es gibt nun unterschiedliche Möglichkeiten, dieses indirekt gegebene Intervall zu interpretieren.

Im folgenden werden die Stützpunktlisten in Formeln als Menge ihrer Stützpunkte dargestellt:

Definition 2.6

$$\textit{Stützpunktliste } \mathbb{S}_n := \textit{Menge aller Stützpunkte des Performanzprofils } n$$

Die Reihenfolge der Stützpunkte ist nach wie vor durch ihre Ressourcenwerte gegeben.

Zwischen zwei aufeinanderfolgenden Stützpunkten einer solchen Stützpunktliste kann der in diesem Bereich unbekannte Verlauf des Performanzprofils aufgrund der Monotonie auf ein Rechteck beschränkt werden, welches als unteren linken Eckpunkt den ersten und als oberen rechten den zweiten Stützpunkt hat. Es umfaßt somit alle theoretisch möglichen Verläufe des Performanzprofils zwischen den beiden Stützpunkten. Die lineare Strecke zwischen den beiden Punkten teilt die Fläche des Rechtecks in zwei Hälften und stellt somit eine Annäherung des unbekannten Verlaufes durch Mittelung dar.

Diese Interpretation bietet sich an, falls die Stützpunkte auf Messungen der Qualität bei zufälligen und bzw. oder sehr grob gerasterten Ressourcenwerten basieren, und davon ausgegangen werden kann, daß zwischen zwei aufeinanderfolgenden Stützpunkten viele weitere Qualitätsverbesserungen liegen, deren Position nicht weiter bekannt ist.

Sind die Ressourcenwerte der Stützpunkte aber sehr fein gerastert oder es ist bekannt, daß sie genau an Stellen liegen, an denen die Qualität sprunghaft ansteigt, dann ist die Mittelung zwischen zwei aufeinanderfolgenden Stützpunkten eher von Nachteil, denn Anytime-Algorithmen steigern ihr aktuelles Zwischenergebnis, wie in Abschnitt 2.3.1 beschrieben, nicht kontinuierlich, sondern in Sprüngen, z. B. durch Iterationsschritte. Dies kann dazu führen, daß einem Anytime-Algorithmus eine

[6] Dies kann eintreten, falls zwei aufeinanderfolgende Stützpunkte denselben q-Wert haben. In diesem Fall haben aufgrund der Monotonie eines Performanzprofiles alle dazwischenliegenden Punkte denselben q-Wert.

Menge an Ressourcen zugeteilt wird, von der ein Teil für einen Iterationsschritt verwendet wird, der nicht zu Ende geführt werden kann und deswegen auch keine Qualitätsverbesserung mit sich bringt. Auf diese Weise wird ein Teil der verfügbaren Ressourcen verschwendet, der vielleicht an anderer Stelle noch hätte sinnvoll eingesetzt werden können.

Im Fall, daß die Stützpunkte eines Performanzprofils jeweils einer Stufe einer treppenstufenförmigen Qualitätskurve entsprechen, bietet sich eine andere Interpretationsmethode an, die in Abschnitt 4.3.2 näher erläutert wird.

2.5.2. Funktionsparameter

Performanzprofile können häufig ohne große Fehler durch geschlossene Darstellungen von Funktionen geringer Komplexität angenähert werden. Werden die Performanzprofile anstatt als Stützpunktlisten als Funktionsklassentyp mit den dazugehörigen Parametern gespeichert, so kann — je nachdem, wie viele Stützpunkte die entsprechende Stützpunktliste hätte — gegenüber der obengenannten Darstellung in Form von Stützpunktlisten teilweise viel Speicherplatz gespart werden.

Wird allerdings für die Approxmierung eine Funktionsklasse gewählt, die dem gegebenen Performanzprofil nicht sehr ähnelt, so kann es zu größeren Abweichungen zwischen den ursprünglichen und den gespeicherten Performanzdaten [Zilberstein, 1993] kommen.

Im folgenden werden kurz die lineare und die exponentielle Regression zur Annäherung an lineare bzw. exponentielle Funktionen vorgestellt.

2.5.2.1. Lineare Funktionen

Im Normallfall werden lineare Funktionen durch zwei Parameter dargestellt: Steigung und y-Achsenabschnitt bzw. im Fall von Performanzprofilen Steigung und q-Achsenabschnitt.

Um aus Stützpunktlisten — z. B. erzeugt durch die in Abschnitt 2.4 erwähnten Methoden — angenäherte, lineare Funktionen bzw. deren Parameter zu approximieren, kann die lineare Regression (vergleiche Abbildung 2.4), wie sie z. B. in [Bronstein & Semendjajew, 1967] beschrieben ist, verwendet werden:

Sind r_i die Ressourcenwerte eines Performanzprofile und q_i die entsprechenden Qualitätswerte, so lassen sich die Steigung m und der q-Achsenabschnitt q_0 der Geraden,

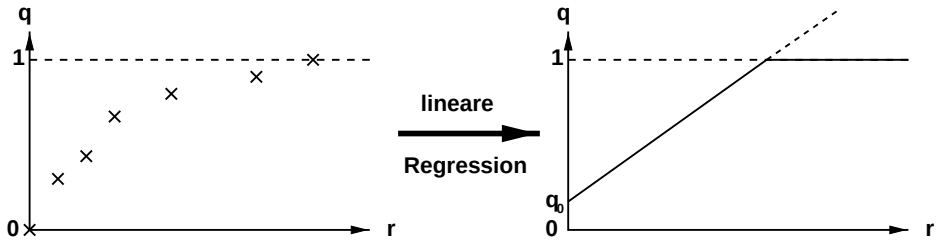

Abbildung 2.4.: Annäherung durch lineare Regression

deren durchschnittlicher Abstand zu allen Stützpunkten minimal ist, mit folgenden Formeln [Baus & Beckert, 1998] errechnen:

$$q_0 = \frac{\sum_i r_i^2 \sum_i q_i - \sum_i r_i q_i \sum_i r_i}{n \sum_i r_i^2 - \left(\sum_i r_i\right)^2} \tag{2.2}$$

$$m = \frac{n \sum_i r_i q_i - \sum_i r_i \sum_i q_i}{n \sum_i r_i^2 - \left(\sum_i r_i\right)^2} \tag{2.3}$$

2.5.2.2. Exponentialfunktionen

Wie die linearen Funktionen können auch exponentielle Funktionen durch Parameter dargestellt werden. Für Performanzprofile werden meist die Funktionsklassen $\mathrm{q}(r) = 1 - e^{-\lambda r}$ [Dean & Boddy, 1988] und $\mathrm{q}(r) = 1 - \eta e^{-\lambda r}$ [Zilberstein, 1993] verwendet. Im Vergleich zu den linearen Funktionen haben sie den Vorteil, daß sie den Qualitätsverlauf vieler rekursiver Näherungsverfahren besser annähern: Zu Beginn sind die Qualitätsgewinne groß, sobald auch nur wenige Ressourcen investiert werden, aber später sind immer mehr Ressourcen notwendig, um auch nur eine geringe Verbesserung zu erreichen.

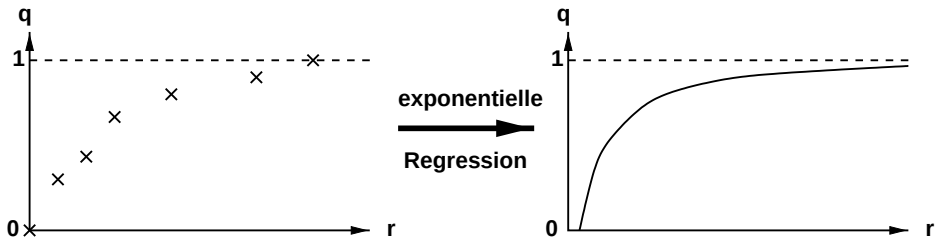

Abbildung 2.5.: Annäherung durch exponentielle Regression

Auch bei diesen Klassen können wieder durch Regression (vergleiche Abbildung 2.5) aus den Stützpunktlisten die Parameter η und λ der Funktion gefunden werden, deren durchschnittlicher Abstand zu allen Stützpunkten minimal ist [Baus & Beckert, 1998]. Dazu wird das Problem auf die lineare Regression (Formeln 2.2 und 2.3) zurückgeführt, indem folgende Äquivalenzumformung betrachtet wird:

$$\begin{array}{rrcl} & q & = & 1 - \eta e^{-\lambda r} \\ \Leftrightarrow & 1 - q & = & \eta e^{-\lambda r} \\ \Leftrightarrow & \ln\left(\frac{1-q}{\eta}\right) & = & -\lambda r \\ \Leftrightarrow & \ln(1-q) & = & \ln(\eta) - \lambda r. \end{array} \tag{2.4}$$

Durch Setzen von $\bar{q} := \ln(1-q)$, $\bar{\eta} := \ln(\eta)$ und $\bar{\lambda} := -\lambda$ ergibt sich eine lineare Funktion:

$$\bar{q} = \bar{\eta} + \bar{\lambda} r \tag{2.5}$$

Durch lineare Regression auf die Punktemenge $\overline{PP} = \{ (r_i, \bar{q_i}) \mid 0 < i \leq |\$_{PP}| \}$ mit $\bar{q_i} = \ln(1 - q_i)$ ergeben sich die Parameter $\bar{\eta}$ und $\bar{\lambda}$ für eine Geradenfunktion, die $\overline{PP}$ annähert und daraus die entsprechenden Parameter $\eta = e^{\bar{\eta}}$ und $\lambda = -\bar{\lambda}$ für die gesuchte exponentielle Funktion.

Dabei ist zu beachten, daß $\bar{q_i} = \ln(1 - q_i)$ für $q_i = 1$ undefiniert ist. Dies ist der Fall, wenn in dem verwendeten Performanzprofil ein Stützpunkt auftritt, dessen Qualitätswert gleich 1 ist.

Exponentielle Funktionen von obiger Form erreichen nie die Gerade $q = 1$, sie nähern sich ihr nur asymptotisch für $r \to \infty$. Daher ist es nicht möglich, Performanzprofile, die diese Gerade erreichen, exakt nachzubilden. Eine sehr pragmatische Lösung, um das Problem der Undefiniertheit von $\bar{q_i}$ für $q_i = 1$ zu umgehen, ist, alle Qualitätswerte des ursprünglichen Performanzprofils mit einen Faktor $1 - \epsilon$ (mit geeignetem, sehr kleinem ϵ) zu multiplizieren, so daß in dem Performanzprofil keine Stützstelle mit $q_i = 1$ mehr vorkommt.

2.5.3. Diskrete Performanzverteilungsprofile

Eine weitere Art von Performanzprofilen sind Performanzverteilungsprofile [Zilberstein, 1993]. Diese geben an, wie wahrscheinlich es ist, daß eine bestimmte Qualität bei einer bestimmten Ressourcenmenge erreicht wird. Sie entsprechen also einer Funktion

$$\text{w} : \text{Ressourcenmenge} \times \text{Qualität} \longmapsto \text{Wahrscheinlichkeit} \quad (2.6)$$
$$\text{mit Ressourcenmenge} \in \mathbb{R}_0^+ \text{ und Qualität,Wahrscheinlichkeit} \in [0;1]\,.$$

Performanzverteilungsprofile werden meist in Tabellen wie Tabelle 2.1 gespeichert. In diesem Beispiel ist die Wahrscheinlichkeit 0,7, daß nach 5 verbrauchten Ressourceneinheiten die Qualität 60% oder mehr erreicht hat.

Qualität → Ressourcen ↓	0,0	0,2	0,4	0,6	0,8	1,0
0,0	1,00	—	—	—	—	—
1,0	0,70	0,25	0,05	—	—	—
2,0	0,20	0,40	0,30	0,10	—	—
3,0	0,10	0,20	0,40	0,20	0,10	—
4,0	—	0,10	0,30	0,40	0,15	0,05
5,0	—	0,05	0,20	0,45	0,20	0,10
6,0	—	—	0,10	0,25	0,50	0,15
7,0	—	—	—	0,10	0,60	0,30
8,0	—	—	—	—	0,30	0,70
9,0	—	—	—	—	—	1,00

Tabelle 2.1.: Beispiel für ein Performanzverteilungsprofil in Tabellenform

Denkbar wären aber auch andere, genauere Darstellungsformen, z. B. die im folgenden Abschnitt beschriebenen Lambda-Ausdrücken. Dabei müssen die von den Lambda-Ausdrücke dargestellten Funktionen neben der Ressourcenmenge R auch noch die Qualität q als Parameter bekommen und als Ergebnis liefern, wie groß die Wahrscheinlichkeit ist, daß nach r verbrauchten Ressourceneinheiten die Qualität q erreicht wird.

Performanzverteilungsprofile werden in dieser Arbeit nicht weiter behandelt, da die Entwicklungsvorgabe für ORCAN ein System vorsah, welches Performanzprofile in verschiedenen Darstellungsformen von Funktionen der Art q : Ressourcenmenge $\longmapsto$ Qualität als Parameter übergeben bekommt.

2.6. Kompilierung von Anytime-Algorithmen

Bis zu dieser Stelle wurden nur einzelne Anytime-Algorithmen und ihre Repräsentation durch Performanzprofile betrachtet. Viele Aufgaben lassen sich aber nicht

durch einen einzigen Anytime-Algorithmus bewältigen, müssen also aufgrund ihrer Komplexität in Teilaufgaben zerlegt werden. Als Beispiel seien hier das in der Einleitung bereits erwähnte Projekt REAL genannt, das u. a. folgende Teilaufgabe hat: Die Analyse einer Wo-Frage bezüglich der darin enthaltenen Raumbeschreibungen, das Finden eines besten Referenzobjektes und das Generieren einer dazu relativen Raumbeschreibung.

Entsprechend ist die Kombination mehrerer Anytime-Algorithmen zu einem komplexen Anytime-System nötig, und damit stellt sich die Frage nach der Verteilung der jeweils zur Verfügung stehenden Rechenzeit auf die einzelnen Anytime-Algorithmen, so daß die Qualität des Gesamtergebnisses maximiert wird.

Die *Kompilierung von Anytime-Algorithmen nach Zilberstein (1993)* beschreibt einen Ansatz für das Umwandeln eines (nicht direkt ausführbaren) *zusammengesetzten Anytime-Moduls*, das aus mehreren Anytime-Algorithmen besteht, zu einem *ausführbaren Anytime-Modul*. Dieses beinhaltet ein *kompiliertes zusammengesetztes Anytime-Modul*, ein System-Performanzprofil und einen *Monitor*, welcher für die Überwachung der einzelnen Anytime-Algorithmen innerhalb des Anytime-Moduls zuständig ist. Die Berechnung der Ressourcenverteilungen geschieht dabei vor dem Einsatz des Anytime-Moduls durch einen Kompiler, welcher als Eingabe das *zusammengesetzte Anytime-Modul* und die Performanzprofile der darin enthaltenen Anytime-Algorithmen bekommt. Die Ergebnisse dieser Berechnung werden dann dem Anytime-Modul in Form eines System-Performanzprofils mitgegeben. Dieser Ablauf ist in Abbildung 2.6 schematisch dargestellt.

Ein auf diese Weise generiertes Anytime-Modul besitzt die Eigenschaften eines Contract-Algorithmus und kann damit nach Satz 2.1 auch in einen echten Anytime-Algorithmus umgewandelt werden.

Weiter muß bei der Kompilierung darauf geachtet werden, ob und – falls ja – wie sich die verschiedenen Anytime-Algorithmen gegenseitig beeinflussen, also z. B. ob das Ergebnis eines Anytime-Algorithmus als Eingabe eines anderen verwendet wird. Diese Abhängigkeiten werden dem Kompiler in Form einer n-stelligen Verknüpfungsfunktion

$$\mathrm{Q} : \mathrm{Qualität}^n \longmapsto \mathrm{Qualität} \tag{2.7}$$

übergeben, wobei n die Anzahl der zu kompilierenden Anytime-Algorithmen ist. Sie verknüpft die zu erwartenden Qualitäten der einzelnen Anytime-Algorithmen entsprechend ihrer Verhältnisse untereinander zur zu erwartenden Gesamtqualität.

Sind z. B. Ressourcen auf drei Anytime-Algorithmen zu verteilen und das Ergebnis

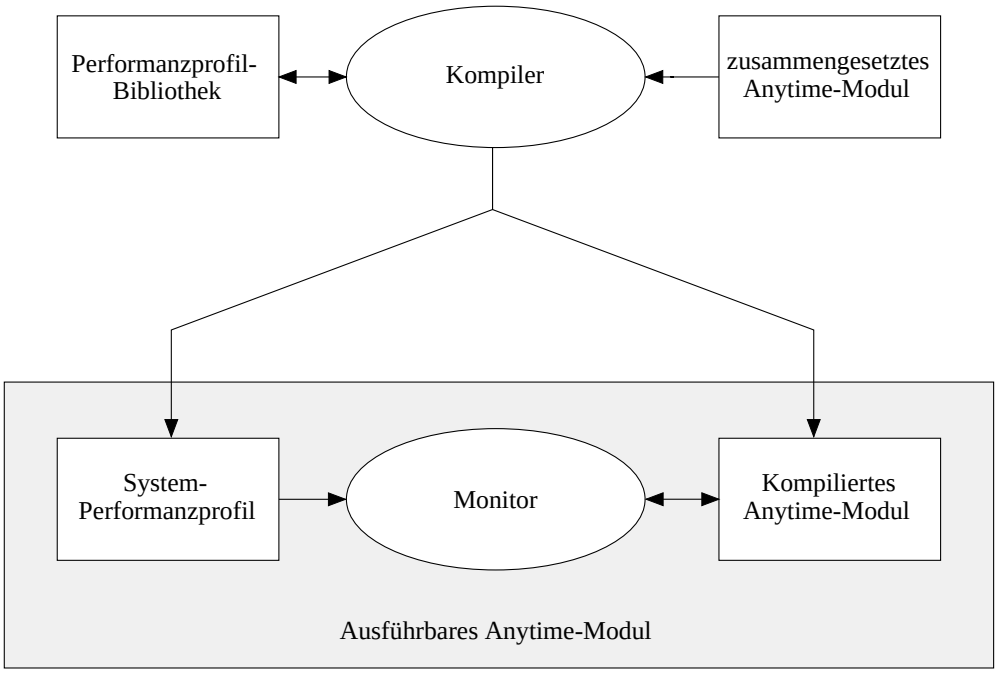

Abbildung 2.6.: Kompilierung nach Zilberstein (1993, Seite 56)

der ersten Anytime-Algorithmen dient dem zweiten als Eingabe, so ist als Verknüpfungsfunktion, die die Verhältnisse der drei Anytime-Algorithmen untereinander darstellt und ihre Gesamtqualität zurückliefert, die Funktion

$$\mathrm{Q}(q_1, q_2, q_3) = (q_1 \cdot q_2) + q_3 \tag{2.8}$$

geeignet.

Sind alle Anytime-Algorithmen voneinander unabhängig, d. h. kein Ergebnis eines der Anytime-Algorithmen dient einem anderen als Eingabe, so ist z. B. die normale Addition oder die Multiplikation eine Verknüpfungsfunktion, die die Verhältnisse aller zu kompilierenden Anytime-Algorithmen untereinander darstellt und deren Gesamtqualität zurückliefert. Es können aber auch komplexere Funktionen für diesen Fall verwendet werden, wie beispielsweise die Funktion

$$\mathrm{Q}\left(q_1, \ldots, q_n\right) = \frac{q_1 + \cdots + q_n + \left(q_1 \cdot \cdots \cdot q_n \cdot n\right)}{n+1}. \tag{2.9}$$

Der dazugehörige Common-Lisp-Quelltext findet sich auf Seite 175.

2.6.1. Lokale Kompilierung

Die im vorherigen Abschnitt vorgestellte Kompilierung birgt ein großes Problem: Die Komplexität dieses Optimierungsproblems steigt exponentiell mit der Anzahl der zu kompilierenden Anytime-Algorithmen.

Um dieses Problem zu umgehen, schlägt Zilberstein (1993) vor, die bisher globale Optimierung der Ressourcenverteilung – also die Verteilung der Ressourcen auf alle Anytime-Algorithmen gleichzeitig – durch eine lokale Optimierung zu ersetzen, die jeweils die Ressourcen nur auf die direkten Subkomponenten verteilt.

Dazu werden alle vorhandenen Ressourcen auf sämtliche direkten Subkomponenten verteilt, welche die ihnen zugeteilten Ressourcen dann rekursiv auf ihre Subkomponenten weiterverteilen.

2.6.2. Gesamtprofile

Um aber Ressourcen auf die direkten Subkomponenten verteilen zu können, müssen zuerst einmal Performanzprofile für diese erzeugt werden. Das Performanzprofil einer Subkomponente ensteht durch Kompilierung der Performanzprofile der eigenen Subkomponenten, welche wiederum ihre Performanzprofile durch die rekursive Kompilierung all ihrer Subkomponenten erhalten. Diese Performanzprofile von Subkomponenten werden in [Baus & Beckert, 1998] *Gesamtprofile* genannt, da sie ein Performanzprofil für das gesamte Element inklusive all seiner Subkomponenten darstellt.

D. h. die baumartige Hierarchie der Komponenten wird mehrmals durchgangen: Im ersten Durchlauf werden von den Blättern her alle Performanzprofile der Subkomponenten berechnet, und danach werden von der Wurzel aus die Ressourcen anhand der Gesamtprofile auf die Subkomponenten verteilt. Diese Hierarchie von Komponenten entspricht im Normalfall der Hierarchie der Anytime-Algorithmen innerhalb des Systems.

2.6.3. Binärbäume

In [Baus & Beckert, 1998] wird eine weitere Anwendung der lokalen Kompilierung gezeigt: Bei Kompilierungsmethoden, welche die als Parameter übergebenen Ressourcen nur auf maximal zwei Performanzprofile gleichzeitig verteilen können, können durch lokale Kompilierung mit den Performanzprofilen als Blätter auch Ressourcen

auf mehr als zwei Anytime-Algorithmen verteilt werden. Dazu werden die vorhandenen Performanzprofile in zwei Hälften geteilt und jede der Hälften für sich kompiliert und danach deren Gesamtprofil kompiliert. Dies geschieht rekursiv, so daß jeder Knoten im Hierachiebaum nur zwei Kinder hat.

Zwei solche, auf Beispielen in [Zilberstein, 1993] basierende Kompilierungsmethoden werden in den Abschnitten 3.1.1 und folgende näher erläutert.

3. Methoden zur Berechnung von Ressourcenverteilungen

In diesem Kapitel werden verschiedene Methoden zur Berechnung von Ressourcenverteilungen beschrieben. Diese können von einem Kompiler nach Zilberstein (1993) genutzt werden, sind aber auch völlig unabhängig davon zur Berechnung von Ressourcenverteilungen nutzbar.

Nach Zilberstein (1993) ist die Anzahl der möglichen Ressourcenverteilungen

$$|\mathbb{V}| = \frac{(r + |\mathbb{P}| - 1)!}{r! \cdot (|\mathbb{P}| - 1)!}, \tag{3.1}$$

wobei die Einheit von r die kleinstmögliche Ressourcenmenge ist, in die die verwendete Ressourcenart aufgeteilt werden kann. Die Größe von $\mathbb{V}$ steigt sowohl mit t als auch n exponentiell. Kann die verwendete Ressource in beliebig kleine Teile unterteilt werden, so ist $\mathbb{V}$ unendlich groß. Werden anstatt Performanzprofilen Performanzverteilungsprofile verwendet, so wird das Problem nochmals rechenaufwendiger.

Diese Komplexität kommt vor allem bei nicht-approximativen Kompilierungsmethoden, wie dem Lösen von Differentialgleichungen, zu tragen.

3.1. Lösen von Differentialgleichungen

Ist die n-dimensionale Verknüpfungsfunktion

$$Q\left(\mathrm{q}_1\left(r_1\right), \ldots, \mathrm{q}_n\left(r_n\right)\right) \qquad \text{mit} \quad \mathrm{q}_1, \ldots, \mathrm{q}_n \quad \text{Performanzprofilfunktionen} \tag{3.2}$$

differenzierbar, so ist das Finden des globalen Maximums dieser Funktion (in Abhängigkeit von $r_{\text{gesamt}} = \sum_{i=1}^{n} r_i$) mit Hilfe der Differentialrechnung eine Möglichkeit zum Berechnen einer Ressourcenverteilung auf n Anytime-Algorithmen. Dazu muß die Ableitung dieser Funktion $Q\left(\mathrm{q}_1\left(r_1\right), \ldots, \mathrm{q}_n\left(r_n\right)\right) = 0$ gesetzt werden, woraus sich ein System aus $n-1$ zu lösenden Differentialgleichungen ergibt:

$$\begin{aligned} \frac{\delta Q}{\delta r_1} &= 0 \\ &\vdots \\ \frac{\delta Q}{\delta r_{n-1}} &= 0 \end{aligned} \tag{3.3}$$

Zilberstein (1993) zeigt zwei Beispiele für die Berechnung von Ressourcenverteilungen auf zwei Anytime-Algorithmen in Abhängigkeit von r_{gesamt} durch das Lösen von Differentialgleichungen. Die beiden Beispiele basieren auf linearen und exponentiellen Funktionen, da diese Funktionsfamilien leicht zu differenzieren sind.

Darauf aufbauend haben Baus und Beckert (1998) zwei Kompilierungsmethoden für eine beliebige Anzahl von Performanzprofile entwickelt, welche die Komplexität durch Zuhilfenahme der lokalen Kompilierung reduzieren. Sie werden in den beiden folgenden Abschnitten vorgestellt.

3.1.1. Kompilierung zweier linearer Performanzprofile

Im folgenden wird die Kompilierung zweier durch Performanzprofile in Form linearer Funktionen repräsentierter Anytime-Algorithmen durch das Lösen von Gleichungen beschrieben. Nachdem die Performanzprofile PP_1 und PP_2 – gegebenenfalls durch lineare Regression wie in Abschnitt 2.5.2.1 beschrieben – in die Form

$$\mathrm{q}_{PP_1}(r) = m_1 r + q_1 \tag{3.4}$$
$$\mathrm{q}_{PP_2}(r) = m_2 r + q_2 \tag{3.5}$$

mit den Steigungen m_1 und m_2 sowie den q-Achsenabschnitten q_1 und q_2 gebracht wurden, können die zu vergebenden Ressourcen r_{gesamt} wie folgt auf die beiden Anytime-Algorithmen verteilt werden [Zilberstein, 1993; Baus & Beckert, 1998]:

$$r_{PP_1}(r_{\text{gesamt}}) = \frac{r_{\text{gesamt}}}{2} + \frac{m_1 q_2 - m_2 q_1}{2 m_1 m_2} \tag{3.6}$$

$$r_{PP_2}(r_{\text{gesamt}}) = \frac{r_{\text{gesamt}}}{2} + \frac{m_2 q_1 - m_1 q_2}{2 m_1 m_2} \tag{3.7}$$

Das Gesamtprofil der beiden Performanzprofile ist dementsprechend die Funktion

$$\begin{aligned} \mathrm{q}_{ges}^{lin}(r_{\text{gesamt}}) &:= \mathrm{q}_{PP_1}(r_{PP_1}(r_{\text{gesamt}})) \cdot \mathrm{q}_{PP_2}(r_{PP_2}(r_{\text{gesamt}})) \\ &= \left(q_1 + m_1 \left(\frac{r_{\text{gesamt}}}{2} + \frac{m_1 q_2 - m_2 q_1}{2 m_1 m_2}\right)\right) \\ &\quad \cdot \left(q_2 + m_2 \left(\frac{r_{\text{gesamt}}}{2} + \frac{m_2 q_1 - m_1 q_2}{2 m_1 m_2}\right)\right) \\ &= \frac{(m_1 m_2 r_{\text{gesamt}} + m_1 q_2 + m_2 q_1)^2}{4 m_1 m_2}, \end{aligned} \tag{3.8}$$

welche nur noch von r_{gesamt} abhängig ist.

3.1.2. Kompilierung zweier exponentieller Performanzprofile

Die Kompilierung zweier durch Performanzprofile der Art $\mathrm{q}(r) = 1 - \eta e^{-\lambda r}$ repräsentierte Anytime-Algorithmen durch das Lösen von Gleichungen funktioniert analog zur Variante mit linearen Funktionen. Nachdem die Performanzprofile PP_1 und PP_2 – gegebenenfalls durch exponentielle Regression wie in Abschnitt 2.5.2.2 beschrieben – in die Form

$$\mathrm{q}_{PP_1}(r) = 1 - \eta_1 e^{-\lambda_1 r} \tag{3.9}$$

$$\mathrm{q}_{PP_2}(r) = 1 - \eta_2 e^{-\lambda_2 r} \tag{3.10}$$

gebracht wurden, können die zu vergebenden Ressourcen r_{gesamt} wie folgt auf die beiden Anytime-Algorithmen verteilt werden [Baus & Beckert, 1998]:

$$r_{PP_1}(r_{\text{gesamt}}) = \frac{\ln(\eta_1 \lambda_1) - \ln(\eta_2 \lambda_2) + \lambda_2 r_{\text{gesamt}}}{\lambda_1 + \lambda_2} \tag{3.11}$$

$$r_{PP_2}(r_{\text{gesamt}}) = \frac{\ln(\eta_2 \lambda_2) - \ln(\eta_1 \lambda_1) + \lambda_1 r_{\text{gesamt}}}{\lambda_1 + \lambda_2} \tag{3.12}$$

Das Gesamtprofil der beiden Performanzprofile wird ebenfalls analog zur linearen Variante gebildet und ist ebenfalls nur noch von r_{gesamt} abhängig:

$$\begin{aligned} \mathrm{q}_{ges}^{exp}(r_{\text{gesamt}}) &:= \mathrm{q}_{PP_1}(r_{PP_1}(r_{\text{gesamt}})) \cdot \mathrm{q}_{PP_2}(r_{PP_2}(r_{\text{gesamt}})) \\ &= 2 - \eta_1 e^{-\lambda_1 \cdot \frac{\ln(\eta_1\lambda_1) - \ln(\eta_2\lambda_2) + \lambda_2 r_{\text{gesamt}}}{\lambda_1+\lambda_2}} \\ &\quad - \eta_2 e^{-\lambda_2 \cdot \frac{\ln(\eta_2\lambda_2) - \ln(\eta_1\lambda_1) + \lambda_1 r_{\text{gesamt}}}{\lambda_1+\lambda_2}} \end{aligned} \tag{3.13}$$

3.1.3. Kombination von mehr als zwei Performanzprofilen

Sollen mehr als zwei Performanzprofile mit einer dieser beiden Methoden kompiliert werden, so wird die Menge der Performanzprofile in zwei Hälften zerlegt und jede der beiden getrennt kompiliert, gegebenenfalls wieder durch Halbierung und Kompilierung der beiden Hälften, wie bereits im Abschnitt über lokale Kompilierung (2.6.1) beschrieben.

Dieser Ablauf ist in Algorithmus 1 nochmals als Pseudocode dargestellt. Dabei sind $PP_1, \ldots, PP_n$ die zu kompilierenden Performanzprofile.

Algorithmus 1 `Gesamtprofil_n` $(\mathtt{PP_1}, \ldots, \mathtt{PP_n})$

1: **if** $n = 1$ **then {**
2: Gebe PP_1 zurück.
3: **} elsif** $n = 2$ **then {**
4: Berechne das Gesamtprofil von PP_1 und PP_2 nach Formel 3.8 bzw. 3.13 und gebe es zurück.
5: **} else {**
6: Teile $\{PP_1, \ldots, PP_n\}$ in zwei Mengen

$$\mathbb{G}_1 \leftarrow \left\{PP_1, \ldots, PP_{\lfloor \frac{n}{2} \rfloor}\right\} \tag{3.14}$$

$$\mathbb{G}_2 \leftarrow \left\{PP_{\lceil \frac{n}{2} \rceil}, \ldots, PP_n\right\} \tag{3.15}$$

und gebe `Gesamtprofil_n` $(\mathtt{Gesamtprofil\_n}(\mathbb{G}_1), \mathtt{Gesamtprofil\_n}(\mathbb{G}_2))$ zurück.
7: **}** /* if */

3.2. Hillclimbing-Algorithmen

Zilberstein (1993, Seite 68 ff.) zeigt, daß die Kompilierung von Anytime-Algorithmus NP-vollständig ist. Deshalb bietet es sich vor allem bei hohen Anzahlen von zu kompilierenden Anytime-Algorithmen an, Näherungsverfahren zu verwenden, welche akzeptable Lösungen bei im Vergleich zu den exakten Verfahren wesentlich geringerem Ressourcenverbrauch erreichen.

Als eine einfach zu implementierende Methode hierfür sind Hillclimbing-Algorithmen — auch als *Austauschheuristiken*[1] oder *Lokale Suchen*[2] — bekannt.

3.2.1. Grundgedanke

Ein Hillclimbing-Algorithmus sucht eine Funktion nach einem Extremum ab, indem er von einem Startpunkt[3] aus immer eine bestimmte Schrittweite in die Richtung weitergeht, in der ein besserer (z. B. bei der Suche nach einem Maximum ein höherer) Funktionswert zu finden ist. Eine einfachere Variante ändert die Richtung erst dann, wenn der Funktionswert wieder schlechter wird. Ist irgendwann kein besserer Wert mehr zu finden, wird die Schrittweite herabgesetzt und weitergesucht, bis eine bestimmter Grenzwert (z. B. bezüglich der Schrittweite oder der Qualitätsverbesserung) unterschritten wurde.

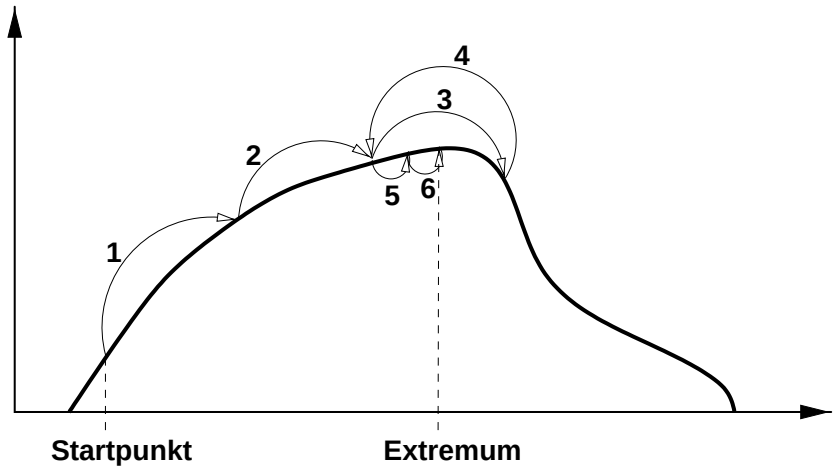

Abbildung 3.1.: Skizze des Ablaufes eines Hillclimbing-Algorithmus

[1] *engl.* Interchange Heuristics [Murty, 1995]
[2] *engl.* Local Search Heuristics [Murty, 1995]
[3] Die Funktionsparameter werden als Vektoren betrachtet.

Abbruchbedingungen von Hillclimbing-Algorithmen sind meist untere Toleranzgrenzen für Schrittweite, Fehlerabweichung oder Funktionswertänderung. Abbildung 3.1 zeigt den Ablauf eines Hillclimbing-Algorithmus zur Bestimmung eines Maximums auf einer einstelligen Funktion.

Allgemeiner Nachteil dieser Vorgehensweise ist die Möglichkeit, nur ein lokales Extremum zu finden, das gegebenenfalls sehr weit vom globalen Extremum entfernt ist. Vorteil der Hillclimbing-Algorithmen ist, daß auf einfache Weise Ressourcen gleichzeitig auch auf mehr als zwei Module verteilt werden können. Dazu wird ein Hillclimbing-Algorithmus auf eine Verknüpfungsfunktion, wie in Formel 2.7 beschrieben angesetzt.

Zur Veranschaulichung eines Hillclimbing-Algorithmus eignet sich sehr gut die Variante mit zwei Performanzprofilen (siehe Abbildung 3.2). In der dritten Dimension wird dann die zu erwartende Qualität

$$\mathrm{Q}\left(r_1, r_2\right) = \mathrm{q}_1\left(r_1\right) \circ \mathrm{q}_2\left(r_{\text{gesamt}} - r_1\right) \tag{3.16}$$

aufgetragen. (Abbildung 3.2b)

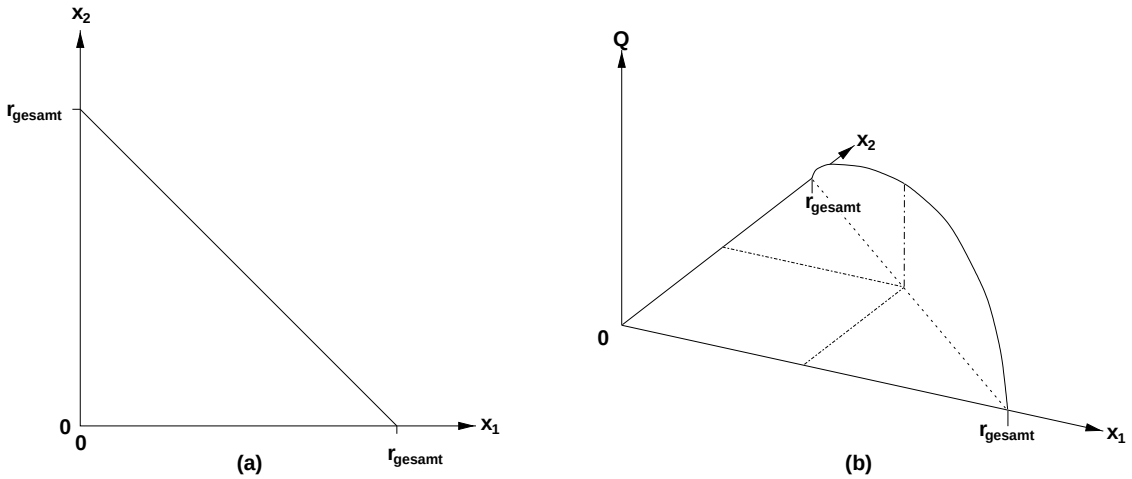

Abbildung 3.2.: Zweidimensionaler Hillclimbing-Algorithmus

Der Algorithmus bewegt sich dabei ausschließlich auf der Strecke zwischen den beiden Punkten $(0 \mid r_{\text{gesamt}})$ und $(r_{\text{gesamt}} \mid 0)$ (Abbildung 3.2a). Deshalb gilt

$$r_2 = r_{\text{gesamt}} - r_1 \quad \text{mit} \quad r_1, r_2 > 0 \tag{3.17}$$

und damit ist die Formel 3.16 bei festem r_{gesamt} auf

$$\mathrm{Q}\left(r_1\right) = PP_1\left(r_1\right) \circ PP_2\left(r_{\text{gesamt}} - r_1\right) \tag{3.18}$$

reduzierbar.

Aus dem zweidimensionalen Modell aus Formel (3.16) kann das n-dimensionale Modell für

$$\mathrm{Q}\left(\mathrm{q}_1, \ldots, \mathrm{q}_n\right) = \mathrm{q}_1\left(r_1\right) \circ \mathrm{q}_2\left(r_2\right) \circ \cdots \circ \mathrm{q}_n\left(r_n\right) \tag{3.19}$$

abgeleitet werden, wobei immer

$$r_{\text{gesamt}} = \sum_{i=1}^{n} r_i \tag{3.20}$$

gilt.

Bei Darstellung des n-dimensionalen Modells mit $n = 3$ bietet es sich an, sich die Qualität als Dichte im Raum vorzustellen (siehe Abbildung 3.3b, Darstellung der Qualität in Form von Isographen). Der Hillclimbing-Algorithmus selbst läuft nun nicht entlang einer Geraden, sondern auf einer Ebene. Allerdings ist der mögliche Raum für Lösungen durch die Gleichung 3.20 und die Bedingung, daß keine der Ressourcenmengen negative Werte haben darf, auf eine bestimmte Fläche beschränkt (siehe Abbildung 3.3a), so wie der Lösungsraum beim zweidimensionalen Hillclimbing-Algorithmus auf eine Strecke begrenzt wurde (siehe Abbildung 3.2a).

3.2.2. Startwerte

Als Standardwerte für den Start des Hillclimbing-Algorithmus wird — sofern kein Vorwissen über bessere Verteilungen vorhanden ist — meist[4] die Gleichverteilung der Ressourcen gewählt:

$$r_i = \frac{r_{\text{gesamt}}}{|\mathbb{P}|} \; \forall i \in \{1; \ldots; |\mathbb{P}|\} \tag{3.21}$$

[4] so auch in ORCAN

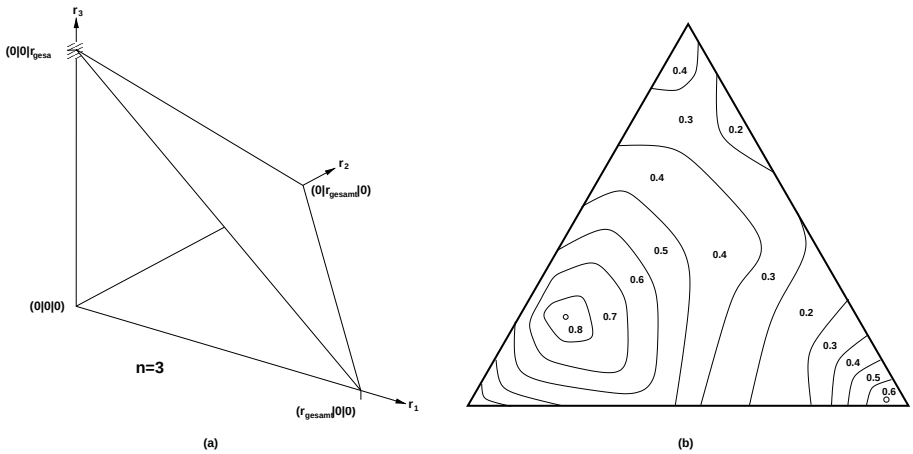

Abbildung 3.3.: Dreidimensionaler Hillclimbing-Algorithmus

Da

$$\sum_{i=1}^{|\mathbb{P}|} \frac{r_{\text{gesamt}}}{|\mathbb{P}|} = r_{\text{gesamt}} \tag{3.22}$$

liegt der Punkt $\left(\frac{r_{\text{gesamt}}}{|\mathbb{P}|} \,\middle|\, \cdots \,\middle|\, \frac{r_{\text{gesamt}}}{|\mathbb{P}|} \right)$ im Ergebnisraum.

In den folgenden beiden Abschnitten wird näher auf zwei unterschiedliche Hillclimbing-Algorithmen zur Ressourcenverteilung eingegangen.

3.2.3. Hillclimbing-Algorithmus nach Zilberstein

Der Hillclimbing-Algorithmus, wie er in [Zilberstein, 1993, Seite 79] beschrieben ist, basiert auf der Idee, daß durch Verschieben von Ressourcen von einem Modul zu einem anderen die Qualität verbessert werden kann. D. h. er verschiebt solange Ressourcen zwischen jeweils den beiden Modulen hin und her, die bei einem Austausch von Ressourcen die größte Qualitätsteigerung mit sich bringen, bis es kein Paar von Modulen mehr gibt, bei denen eine Ressourcenverschiebung zwischen ihnen eine Qualitätsverbesserung mit sich bringt.

3.2.4. Hillclimbing-Algorithmus nach Baus & Beckert

Der von Baus und Beckert (1998) beschriebene Hillclimbing-Algorithmus fängt wie der von Zilberstein (1993) mit einer gleichverteilten[5] Aufteilung an und verteilt die zu vergebende Gesamtzeit solange um, bis ein Optimum mit einer vorgegebenen Genauigkeit gefunden wird. Die Schrittweite entspricht hier der umzuverteilenden Menge an Ressourcen. Der Hillclimbing-Algorithmus rechnet zuerst für jedes Performanzprofil[6] aus, was für eine Qualitätsänderung sich bei einer Ressourcenverteilung zugunsten des entsprechenden Performanzprofils ergeben würde und wählt daraus die günstigste Richtung aus, also jene mit der größten Qualitätssteigerung.

Ist der Algorithmus an einer Stelle angelangt, an der die aktuell eingeschlagene Richtung keine Verbesserung mit sich bringt, so werden von diesem Punkt aus alle anderen Richtungen ausgetestet und die beste davon ausgesucht. Findet der Algorithmus keine Verbesserung in irgendeine Richtung, so wird die Schrittweite heruntergesetzt und erneut alle Richtungen nach der besten Möglichkeit abgesucht.

3.2.4.1. Abbruchbedingungen

Abbruchbedingungen für die Suche sind bei Baus und Beckert (1998):

- Eines der Performanzprofile hat die gesamte Zeit zugewiesen bekommen.
- Die Schrittweite und (bzw. je nach Parameter auch oder) Qualitätsänderung haben ihre Toleranzgrenze erreicht, d. h. ein Optimum ist mit ausreichender Genauigkeit gefunden worden.

Bei einer Implementation als Anytime-Algorithmus kommt an dieser Stelle noch der Abbruch durch das Scheduling-System des übergeordneten Anytime-Moduls dazu.

3.2.4.2. Umverteilung

Bei der Umverteilung der zu vergebenden Zeit ist wichtig, daß keines der Performanzprofile weniger als Null Ressourceneinheiten – also eine negative Ressourcenmenge

[5] bei [Baus & Beckert, 1998] auch optional durch Parameter bestimmbar

[6] Die Nummer des Performanzprofils wird im folgenden als Richtung bezeichnet. Dies läßt sich in Abbildung 3.3 leicht veranschaulichen: Jede Umverteilung zugunsten eines Moduls bedeutet eine Bewegung des aktuellen „Standpunktes“ innerhalb des Dreiecks in Richtung der entsprechenden Ecke.

– zugewiesen bekommt[7].

Dies wird erreicht, indem die aktuelle prozentuale Ressourcenverteilung aller Performanzprofile, die *keine* Zeit hinzu addiert bekommen sollen, berechnet und als Verteilungsvektor abgespeichert wird. Dann werden entsprechend den Prozentsätzen des Verteilungsvektors die Zeiteinheiten abgezogen. Kurz: Ein Anytime-Algorithmus, dem bisher wenig Zeit zugeteilt wurde, bekommt bei einer Umverteilung weniger Zeit abgezogen als ein Anytime-Algorithmus, welcher bisher viel Zeit zugeteilt bekam.

Ein Schritt zur Umverteilung von *step* Ressourceneinheiten auf das Performanzprofil Nummer *dir* sieht dann wie folgt aus:

$$R_{\text{alt}} := \sum_{i=1}^{|\mathbb{P}|} \begin{cases} 0 & \text{falls } i = dir \\ r_{\text{alt}_i} & \text{sonst} \end{cases} \tag{3.23}$$

$$V := \left(\frac{r_{\text{alt}_1}}{R_{\text{alt}}} \,\middle|\, \cdots \,\middle|\, \frac{r_{\text{alt}_{|\mathbb{P}|}}}{R_{\text{alt}}} \right) \tag{3.24}$$

$$step_{\text{tmp}} := \min\{step; R_{\text{alt}}\} \tag{3.25}$$

$$s_i := \begin{cases} +step_{\text{tmp}} & \text{falls } i = dir \\ -step_{\text{tmp}} \cdot \frac{r_{\text{alt}_i}}{R_{\text{alt}}} & \text{sonst} \end{cases} \quad \forall i \in \{1, \ldots, |\mathbb{P}|\} \tag{3.26}$$

$$r_{\text{neu}_i} := r_{\text{alt}_i} + s_i \quad \forall i \in \{1, \ldots, |\mathbb{P}|\} \tag{3.27}$$

Dabei sind die Vektoren r_{alt} die bisherige und r_{neu} die neue Ressourcenverteilung. In Formel 3.23 wird berechnet, wieviel Ressourcen überhaupt noch zum Umverteilen da sind. Formel 3.24 berechnet den aktuellen Verteilungsvektor und Formel 3.25 die für diese Verteilung zu verwendende Schrittweite. In Formel 3.26 wird unter Zuhilfenahme des Verteilungsvektors V die aktuelle Umverteilung errechnet, und in Formel 3.27 wird schließlich die neue Ressourcenverteilung berechnet.

Werden die Ressourcen zugunsten des entsprechenden Performanzprofils solange umverteilt, wie dadurch noch eine Verbesserung der Qualität erreicht wird, so kann der Rechenaufwand – vor allem bei sehr vielen Performanzprofilen – stark verringert werden. Dies läßt sich wie folgt veranschaulichen: Da die optimale Richtung nicht bei jeder Umverteilung neu berechnet wird, ändert sich der Verteilungsvektor weniger oft und muß auch entsprechend seltener berechnet werden.

[7] In bezug auf die Abbildungen 3.2 und 3.3 bedeutet dies, daß der aktuelle Standpunkt innerhalb der Strecke (Abbildung 3.2) bzw. des Dreiecks (Abbildung 3.3) bleibt

3.3. Zusammenfassung

In diesem Kapitel wurden verschiedene bisher benutzte Methoden zur Berechnung von Ressourcenverteilungen erläutert.

Da das Problem der Kompilierung NP-vollständig ist, haben die exakten Berechnungsmethoden den Nachteil, daß mit steigender Performanzprofilanzahl exponentiell viel Rechenzeit für die Berechnung notwendig ist. Eine Möglichkeit, die Komplexität zu verringern, ist die lokale Kompilierung durch die Erstellung von Gesamtprofilen der Subkomponenten.

Eine andere Möglichkeit ist der Einsatz von Hillclimbing-Algorithmen, welche von einer initialen Ressourcenverteilung ausgehen und dann in der lokalen Umgebung eine bessere suchen. Dabei muß allerdings in Kauf genommen werden, daß bei der Verwendung von Hillclimbing-Algorithmen die Gefahr besteht, nur ein lokales Optimum zu finden.

4. Konzeption eines Systems zur Berechnung von Ressourcenverteilungen

4.1. Einsatzzweck und Anforderungen

Das in dieser Arbeit beschriebene System ORCAN ist für die Berechnung von Ressourcenverteilungen auf Anytime-Algorithmen anhand von Performanzprofilen in ressourcenadaptiven Systemen konzipiert. Sind Meta-Daten über die verschiedenen Kompilierungsmethoden, ihre Parameter und gegebenenfalls Eingabeformate vorhanden, so soll es auch in ressourcenadaptierenden Systemen zum Einsatz kommen können.

ORCAN wurde vor dem Hintergrund entwickelt, daß die zu verteilende Ressource Rechenzeit ist, kann aber ohne weiteres auch für die Verteilung von Ressourcen ähnlicher Arten (z. B. Speicherplatz, Treibstoff, etc.) verwendet werden, sofern entsprechende Performanzprofile vorhanden sind.

Für die Berechnung einer Ressourcenverteilung sind als Parameter einerseits die Performanzprofile von Anytime-Algorithmen — oder andere auf 1 normierte Funktionen q : Ressourcenmenge $\longmapsto$ Qualität mit Ressourcenmenge $\in \mathbb{R}_0^+$ und Qualität $\in [0;1]$ — notwendig und andererseits auch die zu verteilende Ressourcenmenge. Über optionale Parameter kann u. a. die verwendete Kompilierungsmethode gewählt und ihr Verhalten beeinflußt werden.

4.2. Unterschiedliche Eingabeformate

ORCAN V1 [Baus & Beckert, 1998] akzeptierte als Eingabeformat ausschließlich

Stützpunktlisten, welche für die Berechnung in andere Formate umgewandelt werden mußten, z. B. durch Regression oder durch die Umwandlung in Funktionen.

Um das Einsatzspektrum zu vergrößern, aber auch, um aus strukturellen Analysen von Anytime-Algorithmen erzeugte Funktionen oder Funktionsparameter als Performanzprofile zuzulassen, sind in ORCAN V2 weitere Eingabeformate möglich. Je nachdem, welche Kombination aus Eingabeformat und Kompilierungsmethode gewählt wird, kann nun auf die Umwandlung der Eingabedaten verzichtet werden.

4.2.1. Funktionsparameter

Den beiden Kompilierungsmethoden, die in ORCAN V1 aus den Stützpunktlisten durch Regression Funktionsparameter errechneten, können die Funktionsparameter nun auch direkt als Parameter übergeben werden. Für lineare Funktionen sind dies die Steigung m und der q-Achsenabschnitt q_0 (siehe Abschnitt 2.5.2.1), für exponentielle Funktionen der Art $\mathrm{q}(r) = 1 - \eta e^{-\lambda r}$ sind dies η und λ (siehe auch Abschnitt 2.5.2.2).

4.2.2. Allgemeine Funktionen in Form von Lambda-Ausdrücken

Bei vielen Programmiersprachen gibt es eine Möglichkeit, Funktionen als Parameter an andere Funktionen zu übergeben, sei es mit Zeigern, durch das die Funktion repräsentierende Symbol oder als sogenannter Lambda-Ausdruck[1], wie es in Lisp-ähnlichen Sprachen meist geschieht. Aus diesen Grund ist es nicht abwegig, auch über Performanzprofile in Form von „Funktionen“ nachzudenken.

Diese Repräsentationsform ist im Gegensatz zu der in Abschnitt 2.5.2 vorgestellten Reduktion der Performanzprofile auf Funktionsparameter auch dann noch eine Alternative zu den Stützpunktlisten, wenn es sich bei dem zu repräsentierenden Performanzprofil um eine Funktion handelt, welche in einem bestimmten Intervall einer gängigen und einem anderen Intervall einer anderen gängigen Funktionsklasse entspricht. Damit ist die Repräsentation in Form von Funktionen beispielsweise für Performanzprofile vorstellbar, die durch strukturelle Analysen (vergleiche Abschnitt 2.4) entstanden sind. Ebenso ist diese Repräsentationsform gut geeignet, wenn es darum geht, komplexe Verknüpfungen der Anytime-Algorithmen untereinander korrekt wiederzugeben.

[1] In Lisp und vielen lisp-ähnlichen Programmiersprachen ist ein Lambda-Ausdruck eine anonyme Funktionsdefinition. Lambda-Ausdrücke bzw. der Name der Lisp-Funktion `#'lambda` gehen auf das λ-Kalkül von Church (1941) zurück.

ORCAN V2 akzeptiert Common Lisp Lambda-Ausdrücke als Performanzprofile für den Hillclimbing-Algorithmus, da dieser seine Berechnungen auf Lambda-Ausdrükken ausführt und deswegen alle als Parameter übergebenen Performanzprofile in anderen Eingabeformaten in Lambda-Ausdrücke umwandelt. Für andere Kompilierungsmethoden ist dieses Eingabeformat nicht verfügbar, da hierzu die durch den Lambda-Ausdruck dargestellte Funktion auf ihren Verlauf untersucht werden müßte. Es ist aber nicht sinnvoll möglich, das für die Berechnung interessante Wertintervall herauszufinden, da schlecht über ganz $\mathbb{R}$ verteilte Stützpunkte gesammelt werden können.

4.2.3. Performanzverteilungsprofile

Performanzverteilungsprofile wie sie im Abschnitt 2.5.3 beschrieben wurden, wurden nicht implementiert, da die hier implementierten Kompilierungsmethoden auf Performanzprofile der Art q : Ressourcenmenge $\longmapsto$ Qualität aufbauen und dies nicht der Datenstruktur der Performanzverteilungsprofile entspricht.

4.3. Unterschiedliche Berechnungsmethoden

Neben den aus ORCAN V1 übernommenen und verbesserten Methoden zur Ressourcenverteilung (linear- und exponentiell-regressive Methoden, Hillclimbing-Methode und abschnittsweise lineare Methode) wurde eine weitere, stark spezialisierte Kompilierungsmethode, die Treppenstufen-Methode entwickelt.

Im folgenden Abschnitt werden kurz die Unterschiede zwischen den beiden Hillclimbing-Algorithmen nach [Zilberstein, 1993] und [Baus & Beckert, 1998] besprochen. Der darauf folgende Abschnitt befaßt sich genauer mit der Treppenstufen-Methode.

4.3.1. Vergleich der beiden genannten Hillclimbing-Algorithmen

Der wichtigste Unterschied zwischen den Hillclimbing-Algorithmen nach Zilberstein (1993) und Baus und Beckert (1998) ist, daß der Hillclimbing-Algorithmus nach Zilberstein immer nur die Zeit zwischen zwei Performanzprofilen austauscht, egal auf wieviele Performanzprofile die Zeit zu verteilen war. Hierzu werden die beiden

Performanzprofile herausgesucht, bei denen die Ressourcenverteilung von dem ersten zu dem zweiten Modul die größte Qualitätssteigerung mit sich bringt.

In [Baus & Beckert, 1998] wird gezeigt, daß der n-dimensionale Hillclimbing-Algorithmus nach Zilberstein (1993) einen Rechenaufwand von $2\,((n-1)!)$ probeweisen Umverteilungen pro Schritt hat, während der oben beschriebene Hillclimbing-Algorithmus nach Baus und Beckert (1998) nur n probeweise Umverteilungen pro Schritt braucht und somit vor allem bei hohem n wesentlich weniger rechenintensiv ist.

Dazu kommt noch, daß beim Hillclimbing-Algorithmus nach Baus und Beckert (1998) (optional bzw. abschaltbar) bei vielen Schritten die Berechnung der möglichen Umverteilungen wegfallen kann und daß die Abbruchbedingungen dort flexibler sind.

Aus diesem Grund wurde in dieser Arbeit darauf verzichtet, den von Zilberstein (1993) vorgeschlagenen Hillclimbing-Algorithmus ebenfalls zu implementieren.

4.3.2. Das Treppenstufen-Verfahren

4.3.2.1. Treppenstufenförmige Interpretation

Im Unterschied zu der in Abschnitt 2.5.1 beschriebenen linearen Interpretationsmethode können die Stützpunktlisten der Performanzprofile auch so interpretiert werden, daß zwischen zwei aufeinanderfolgenden Stützpunkten die jeweils untere Grenze des möglichen Kurvenverlaufs der zu erwartenden Qualität zur Berechnung verwendet wird. Dies garantiert, daß der wirkliche, nur durch seine obere und untere Grenze beschränkte Verlauf der Kurve nie unterhalb der zur Berechnung verwendeten Kurve verläuft und die zu erwartende Qualität somit nicht überschätzt werden kann, selbst wenn zwischen den beiden betrachteten Stützstellen keine weiteren qualitätssteigernden Zwischenergebnisse erreicht werden. Dies bedeutet, daß die Qualität in der grafischen Darstellung bis zur zweiten Stützstelle waagrecht verläuft, dann bei dieser sprunghaft ansteigt und der ganze Graph dann in seiner Form an eine Treppe mit gegebenenfalls unterschiedlich hohen und unterschiedlich langen Stufen erinnert (siehe auch Abbildung 2.2b).

Ein weiterer Vorteil dieser Interpretationsmethode ist, daß bei der Verteilung der Ressourcen auf die Anytime-Algorithmen nicht jede beliebige Menge an Ressourcen für ein bestimmtes Modul in Frage kommt, sondern nur Verteilungen, bei denen die einem Anytime-Algorithmen zugewiesenen Ressourcen genau dem Ressourcenwert eines seiner Stützpunkte entspricht. Denn die Verteilung einer Ressourcenmenge an einen Anytime-Algorithmus, deren Wert zwischen zwei aufeinanderfolgenden Stützstellen des Anytime-Algorithmus liegt, könnte Ressourcen verschwenden, da es

keinerlei Verbesserung der Qualität innerhalb der Differenz zweier aufeinanderfolgenden Stützpunkt geben muß. Entsprechend lohnt es sich nicht, einem Modul eine Menge an Ressourcen zuzuteilen, welche nicht dem r-Wert eines Stützpunktes des entsprechenden Performanzprofils entspricht.

Diese Einschränkung macht aus dem bisherigen stetigen Optimierungsproblem ein diskretes bzw. kombinatorisches Optimierungsproblem, verringert also die Anzahl der möglichen Lösungen von unendlich vielen auf endlich viele. Somit sind weitere Heuristiken auf das Problem der Ressourcenverteilung anwendbar.

Die Anzahl der möglichen Lösungen entsprechen in unserem Fall der Mächtigkeit der Menge der möglichen Ressourcenverteilungen $\mathbb{V}$:

$$|\mathbb{V}| = \prod_{n=1}^{|\mathbb{P}|} |\mathbb{S}_n| \tag{4.1}$$

Diese Beschränkung der Ressourcenverteilung auf ganze Stücke zwischen den einzelnen Stützpunkten ist ebenfalls von Vorteil, wenn die Berechnung einer Qualitätsverbesserung eine Transaktion ist und nicht unterbrochen werden darf, da dann jeder Stützpunkt den Zeitpunkt und die Qualität nach Ablauf einer bzw. vor Beginn der nächsten Transaktion darstellt. Somit kann garantiert werden, daß bei der eigentlichen Berechnung keine der Transaktionen unterbrochen wird.

Außerdem sind bestimmte Stützpunkte, welche für die oben beschriebene lineare Interpretation notwendige Daten enthalten, bei der treppenstufenförmigen Interpretation obsolet: Die Stützpunktlisten können sehr einfach optimiert werden, indem alle Stützpunkte mit gleichem q-Wert (also mit gleicher Qualität) mit Ausnahme des Punktes $(r_i \mid q_i)$ mit dem niedrigsten r-Wert gelöscht werden, da es für die treppenstufenförmige Interpretation der Stützpunktlisten ausreicht, den ersten Stützpunkt jedes vorkommenden q-Wertes zu verwenden. Dies senkt $|\mathbb{V}|$ aus Formel 4.1 weiter und reduziert damit auch die zur Berechnung einer Ressourcenverteilung notwendige Rechenzeit.

4.3.2.2. Algorithmus

Im folgenden wird ein Algorithmus beschrieben, welcher auf Stützpunktlisten, wie sie in Abschnitt 2.5.1 beschrieben sind, arbeitet und die im vorherigen Abschnitt beschriebene treppenstufenförmige Interpretation verwendet. Dieser Algorithmus beschreibt — wie die beiden in Abschnitt 3.2 erwähnten Hillclimbing-Algorithmen — ein heuristisches Verfahren zum Finden eines Extremums, ist aber im Vergleich zum

in Abschnitt 3.2 beschriebenen Hillclimbing-Algorithmus keine Austauschheuristik, sondern ein sogenanntes Greedy-Verfahren[2] [Murty, 1995], welches bei einer „leeren" Lösung beginnt und einmal getroffene Entscheidungen nicht mehr zurücknimmt oder korrigiert. Im Falle des hier beschriebenen Algorithmus werden einmal vergebene Ressourceneinheiten nicht mehr zurückgenommen. Zu Anfang sind 0 Ressourceneinheiten verteilt. Beides ist bei den oben beschriebenen Hillclimbing-Algorithmen nicht der Fall.

Die Grundidee dieses Algorithmus ist, sich in jedem Rekursionsschritt den (bisher noch nicht erreichten) Stützpunkt zu suchen, der noch im Rahmen der verfügbaren Ressourcen liegt und der die effizienteste Qualitätsverbesserung mit sich bringt, wenn seinem Anytime-Algorithmus soviele Ressourcen zugesprochen werden, wie die r-Differenz zwischen ihm und dem ersten Stützpunkt ($r = 0$) im jeweiligen Performanzprofil beträgt. Unter Effizienz wird im Normalfall das Verhältnis von Qualitätssteigerung zu Ressourcenverteilung verstanden, es könnten aber auch andere Bewertungsmaßstäbe, wie z. B. alleine die Qualitätssteigerung, verwendet werden. Da das Verhältnis von Qualitätssteigerung zu Ressourcenverteilung intuitiv das mit Abstand sinnvollste Kriterium zu sein scheint, wurde dies in ORCAN verwendet.

Ein besonderes Merkmal dieses Algorithmus im Vergleich zu den anderen Kompilierungsmethoden ist auch, daß er unabhängig von der Verknüpfungsfunktion der Qualität ist, da diese erst bei der Berechnung der zu erwartenden Gesamtqualität benötigt wird.

Initialisierung: Für jedes Performanzprofil in $\mathbb{P}$ wird ein Stapel angelegt, der die Stützpunkte des entsprechenden Performanzprofils sortiert nach ihrem r-Wert enthält. Dabei ist jeweils der Stützpunkt mit dem kleinsten r-Wert das oberste Element des Stapels.

Stützpunktlisten sind häufig in Datenstrukturen gespeichert, die die Abarbeitung äquivalent zur *pop*-Funktion eines Stapelspeichers (Stacks) erlauben, so daß dieser Schritt wegfallen kann. Solche Datenstrukturen sind z. B. *arrays of arrays* oder die lisp-typischen *lists of lists*.

In den Abbildungen des Beispiel-Durchlaufs in Abschnitt 4.3.2.3 wird der oberste Stützpunkt eines Stapelspeichers durch eine rote Linie auf Höhe dieses Stützpunktes markiert, die gleichzeitig auch den aktuellen Stand der Ressourcenverteilung für jedes der Performanzprofile anzeigt. Die vor der roten Linie liegenden Stützpunkte eines Performanzprofils wurden bereits vom Stapelspeicher genommen.

[2] von *engl.* „greedy" – gierig, habgierig

Rekursionsschritt: Nun wird zwischen dem jeweils obersten Stützpunkt jedes Stapels und allen darauffolgenden Stützpunkten, deren r-Differenz zum obersten Stützpunkt des jeweiligen Stapels noch im verfügbaren Ressourcenrahmen liegt, die Steigung errechnet und daraus der Stützpunkt mit der insgesamt höchsten Steigung herausgesucht.

Anschließend wird dem dazugehörigen Anytime-Algorithmus die Ressourcenmenge zugesprochen, die für die Qualitätsverbesserung zwischen dem aktuell auf der Stapelspitze liegendem Stützpunkt und dem gewählten Stützpunkt nötig ist, und von der noch zur Verfügung stehenden Ressourcenmenge abgezogen. Von dem Stapel, in dem der gewählte Stützpunkt liegt, werden alle über bzw. vor[3] ihm liegenden Stützpunkte entfernt. Bei einer Implementation des Treppenstufen-Verfahrens als Anytime-Algorithmen liegt ab diesem Punkt ein neues, verbessertes Zwischenergebnis vor.

Haben mehrere Stützpunkte dieselbe Steigung und reicht der aktuelle noch zur Verfügung stehende Ressourcenrahmen nur für die Verteilung auf einen der beiden Stützpunkte, so wird der restliche Verteilungsvorgang für beide Möglichkeiten ausgerechnet und danach die Variante mit der höheren zu erwartenden Qualität gewählt.

Die einfachere Lösung, in einem solchen Fall nach der absoluten Qualitätsverbesserung zu gehen, ist gegebenfalls weniger nah am Optimum und deckt noch nicht den Fall ab, daß auch die absolute Qualitätsverbesserung gleich ist. Würde zwischen zwei gleich großen absoluten Qualitätsverbesserungen zufällig gewählt, so beeinträchtigt dies gegebenenfalls die nachfolgenden Verteilungen, da die nun noch im Stapel der beiden fraglichen Performanzprofile liegenden Stützpunkte sehr unterschiedliche Qualitätsverbesserungen mit sich bringen können. Wird die „falsche“ Möglichkeit gewählt, gibt es im schlimmsten Fall keine weitere Verteilungsmöglichkeit, während eine andere Wahl noch wesentliche Qualitätssteigerungen mit sich gebracht hätte.

Bei dieser Lösung handelt es sich zwar im schlimmsten Fall[4] um eine vollständige Suche, aber auch die Lösung, bei der die absolute Qualitätsverbesserung als Entscheidungsgrundlage verwendet wird, läuft, falls auch die Qualitätsverbesserungen mehrerer Performanzprofile gleich groß sind, auf eine vollständige Suche oder eine zufällige Auswahl hinaus. Da der Fall der gleichen Steigung in einem Bereich, in dem nur noch für eines der zutreffenden Performanzprofile Ressourcen übrig sind, nur sehr selten und auch nur bei sehr großen Schritten oder wenigen zu verteilenden

[3] in bezug auf die grafische Darstellung der Beispiele in Abschnitt 4.3.2.3

[4] Dies ist der Fall, wenn sämtliche Performanzprofile gleich sowie so kurz sind, daß nur einem der Performanzprofile der Zuschlag erteilt werden kann.

Ressourcen auftritt[5], stört dies jedoch nicht sehr, wie auch die in Kapitel 6 und Anhang A aufgeführten Laufzeittests zeigen.

Abbruchbedingung: Der Rekursionsschritt wird solange wiederholt, bis es keinen Stützpunkt mehr gibt, der in dem zur Verfügung stehenden Ressourcenrahmen liegt oder — bei der Ausführung als Anytime-Algorithmus — der Algorithmus das Signal zum Abbrechen bekommt. Ersteres ist in den Beispielen im sechsten Rekursionsschritt der Fall (siehe auch Tabelle 4.4).

Ist die Abbruchbedingung der zu kleine Ressourcenrahmen gewesen, so wird je nach Parameter im Anschluß noch eine eventuelle Mehrverteilung errechnet. Details hierzu in Abschnitt 4.3.2.4.

Aufsummierung: Wird das (Zwischen-) Ergebnis in Form einer reinen Aufteilung der Ressourcen, also eine Liste der Art $(r_1, \ldots, r_n)$ benötigt, so wird abschließend für jeden der betrachteten Anytime-Algorithmen die zugesprochenen Ressourcenmengen aufaddiert und der daraus resultierende Verteilungsvektor zurückgegeben. Dies ist in Tabelle 4.4 in den letzten vier Spalten geschehen.

Pseudocode: In Algorithmus 2 ist oben beschriebener Ablauf nochmals als Pseudocode dargestellt. Dabei ist „Differenz$_r$ (P_1, P_2)“ der Abstand der beiden Stützpunkte P_1, „Steigung (P_1, P_2)“ die Steigung der Geraden durch die beiden Stützpunkte P_1 und P_2 sowie „$Stapel[i]$“ das i-te Element des Stapels, von oben und mit 1 begonnen zu zählen.

4.3.2.3. Beispiel-Durchlauf

Als Beispiel für die Verteilung von Ressourcen mit dem Treppenstufen-Verfahren dienen im folgenden die vier Stützpunktlisten

a. $\{(0 \mid 0); (1 \mid 0{,}8); (5 \mid 1)\}$,

b. $\{(0 \mid 0); (1{,}2 \mid 0{,}4); (4 \mid 0{,}6); (6 \mid 1)\}$,

c. $\{(0 \mid 0); (0{,}5 \mid 0{,}1); (1{,}0 \mid 0{,}24); (1{,}5 \mid 0{,}5); (2 \mid 1)\}$ und

[5] In anderen Fällen sind fast immer genügend Ressourcen für alle zutreffende Performanzprofile vorhanden.

Algorithmus 2
`TreppenstufenVerteilung` $(r_{\text{gesamt}}, \{PP_1, \ldots, PP_n\}, Beschränkung)$

1: $\mathbb{E} \leftarrow \emptyset$ /* *Ergebnismenge* */
2: $r_{\text{übrig}} \leftarrow r_{\text{gesamt}}$
3: **for** i from 1 to n **do** { /* *Initialisierung* */
4: **for** j from length (PP_i) down to 1 **do** {
5: Lege $PP_i[j]$ auf $Stapel_i$.
6: } /* for */
7: } /* for */
8: **repeat** {
9: $\mathbb{M} \leftarrow \emptyset$
10: **for** i from 1 to n **do** { /* *Alle Punkte innerhalb der erlaubten Ressourcenmenge finden* */
11: **for** j from 1 to length $(Stapel_i)$ **do** {
12: **if** $r_{\text{übrig}} \geq \text{Differenz}_r\,(Stapel_i[1], Stapel_i[j])$ **then** {
13: $\mathbb{M} \leftarrow \mathbb{M} \cup \{(i, j)\}$
14: } /* if */
15: } /* for */
16: } /* for */
17: **if** $\mathbb{M} \neq \emptyset$ **then** { /* *Falls Punkte gefunden* */
18: $max \leftarrow 0$
19: $i_{\max} \leftarrow$ undef
20: $j_{\max} \leftarrow$ undef
21: **for all** (i, j) in $\mathbb{M}$ **do** { /* *Maximum finden* */
22: **if** Steigung $(Stapel_i[1], Stapel_i[j]) > max$ **then** {
23: $max \leftarrow$ Steigung $(Stapel_i[1], Stapel_i[j])$
24: $i_{\max} \leftarrow i$
25: $j_{\max} \leftarrow j$
26: } /* if */
27: } /* for */
28: $d \leftarrow \text{Differenz}_r\,(Stapel_{i_{\max}}[1], Stapel_{i_{\max}}[j_{\max}])$
29: $\mathbb{E} \leftarrow \mathbb{E} \cup \{(i_{\max}, d)\}$
30: $r_{\text{übrig}} \leftarrow r_{\text{übrig}} - d$
31: } /* if */
32: } **until** $\mathbb{M} = \emptyset$
33: $\mathbb{E} \leftarrow \mathbb{E} \cup$ Mehrvergabe $(r_{\text{gesamt}}, \{Stapel_1, \ldots, Stapel_n\}, r_{\text{übrig}}, Beschränkung)$
/* *Siehe Abschnitt 4.3.2.4* */
34: **for** i from 1 to n **do** { /* *Ergebnisfelder initialisieren* */
35: $Ergebnis[i] \leftarrow 0$
36: } /* for */
37: **for all** (i, s) in $\mathbb{E}$ **do** { /* *Aufsummieren* */
38: $Ergebnis[i] \leftarrow Ergebnis[i] + s$
39: } /* for */
40: Gebe $Ergebnis$ als Ressourcenverteilung zurück.

d. $\{(0 \mid 0);(0{,}2 \mid 0{,}4);(0{,}5 \mid 0{,}7);(1{,}5 \mid 0{,}9);(5 \mid 1)\}$.

Zur Darstellung der Gesamtqualität wird in diesem Beispiel die Addition[6] als Verknüpfungsfunktion verwendet. Insgesamt sind 5,5 Ressourceneinheiten zu verteilen.

In Abbildung 4.1 finden sich mehrere Dinge: Einerseits sind in ihr die verwendeten Performanzprofile in treppenstufenförmiger Interpretation abgebildet. Andererseits stellt sie auch den Zustand zu Beginn des Algorithmus dar: Die senkrechten roten Linien markieren — wie bereits erwähnt — jeweils den obersten Stützpunkt eines Stapelspeichers. Von diesen Stützpunkten gehen jeweils gestrichelte Linien zu allen folgenden Stützpunkten eines Performanzprofils. An ihnen läßt sich auf einfache Weise die Steigung zwischen den entsprechenden Stützpunkten vergleichen.

Da noch keine Ressourcen verteilt sind, liegen alle roten Linien noch bei $r = 0$ und fallen somit mit den q-Achsen der Koordinatensysteme zusammen.

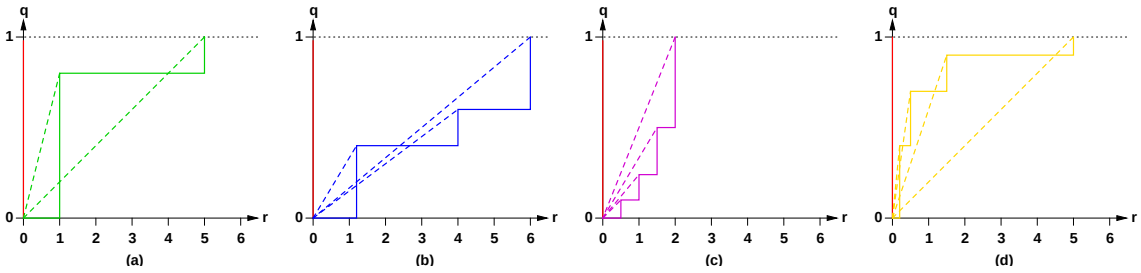

Abbildung 4.1.: Beispiel-Performanzprofile zur Veranschaulichung des Treppenstufen-Verfahrens

Tabelle 4.1 zeigt die Steigungen zwischen dem ersten und den restlichen Stützpunkten eines Stapelspeichers bei einer zu verteilenden Ressourcenmenge von 5,5 Ressourceneinheiten. Ein „—“ bedeutet dabei, daß der Stützpunkt bereits außerhalb der zu vergebenden Ressourcenmenge liegt. Die größte Steigung und der dazugehörige Stützpunkt sind in der Tabelle durch Fettdruck hervorgehoben. Im Beispiel ist das der Stützpunkt $(0{,}2 \mid 0{,}4)$ von Performanzprofil (d) mit einer Steigung von 2.

In folgenden Teil dieses Rekursionsschrittes wurden dem Performanzprofil (d) 0,2 Ressourceneinheiten zugesprochen. Die Gesamtqualität liegt zu diesem Zeitpunkt bei 0,4. Die noch zu verteilende Ressourcenmenge ist auf 5,3 gesunken. Die grafische Darstellung der Ressourcenverteilung und Performanzprofile bzw. Stapelspeicher zu diesem Zeitpunkt ist in Abbildung 4.2 zu sehen. Die Darstellung des Zwischenergebnisses wird analog zu den Performanzprofilen (a) bis (d) ebenfalls in einer Kette von

[6] und nicht das arithmetische Mittel

Performanzprofil (a): Gesamtsteigung des Qualitätsverlaufes vom Stützpunkt (0 | 0) aus.

Stützpunkt:	(1 | 0,8)	(5 | 1)
Steigung:	0,8	0,2

Performanzprofil (b): Gesamtsteigung des Qualitätsverlaufes vom Stützpunkt (0 | 0) aus.

Stützpunkt:	(1,2 | 0,4)	(4 | 0,6)	(6 | 1)
Steigung:	$0,\overline{3}$	0,15	—

Performanzprofil (c): Gesamtsteigung des Qualitätsverlaufes vom Stützpunkt (0 | 0) aus.

Stützpunkt:	(0,5 | 0,1)	(1,0 | 0,24)	(1,5 | 0,5)	(2 | 1)
Steigung:	0,2	0,24	$0,\overline{3}$	0,5

Performanzprofil (d): Gesamtsteigung des Qualitätsverlaufes vom Stützpunkt (0 | 0) aus.

Stützpunkt:	(**0,2** | **0,4**)	(0,5 | 0,7)	(1,5 | 0,9)	(5 | 1)
Steigung:	**2,0**	1,4	0,6	0,2

Tabelle 4.1.: Steigungen der möglichen Schritte bei der Verteilung von 5,5 Ressourcen auf die Beispiel-Performanzprofile im ersten Rekursionsschritt

Treppenstufen sowie im selben Maßstab wie diese dargestellt. Die gestrichelten Linien stellen wieder die Steigung dar. Allerdings sind in der Darstellung des Ergebnisses nur noch die Steigungen eingezeichnet, aufgrund derer auch Ressourcenzuteilungen erfolgten. Zur Veranschaulichung wurde die Gesamtqualität nicht normiert, d. h. die maximal mögliche Qualität ohne Ressourcenbeschränkung liegt bei 4.

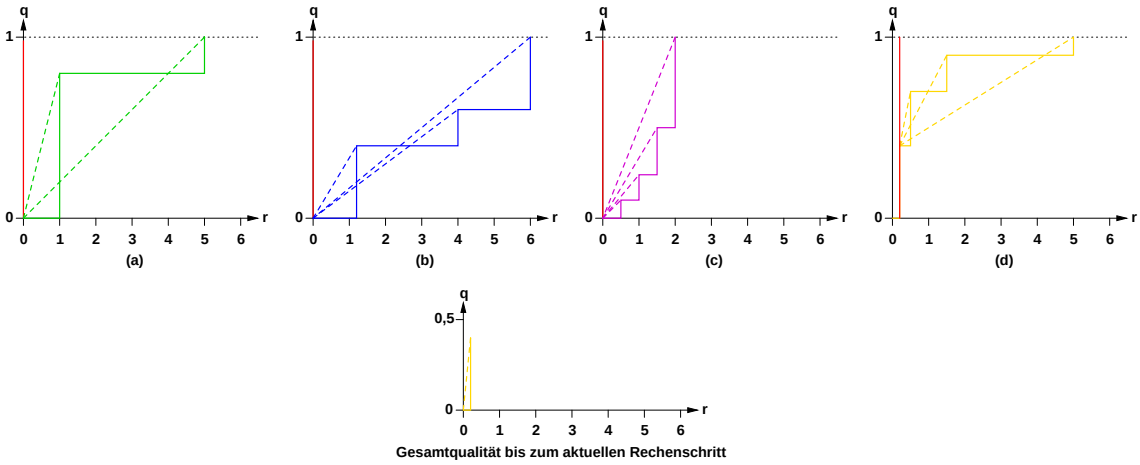

Abbildung 4.2.: Beispiel-Performanzprofile und Zwischenergebnis nach dem ersten Rekursionsschritt

Im nächsten Rekursionschritt werden nur die Steigungen für das Performanzprofil (d) neu berechnet, da sich der Stapelspeicher der anderen Performanzprofile nicht geändert hat. Es muß allerdings überprüft werden, ob Stützpunkte durch die kleiner gewordene Menge an noch verfügbaren Ressourcen aus der Auswahl herausgefallen sind. In diesem Schritt ist dies aber nicht der Fall (siehe Tabelle 4.2). In Abbildung 4.2 sind bereits die neu berechneten Steigungen eingezeichnet.

Im zweiten Schritt wurden dem Performanzprofil (d) weitere 0,3 Ressourceneinheiten zugesprochen, es bekommt jetzt insgesamt 0,5 Ressourceneinheiten. Die Gesamtqualität liegt nun bei 0,7, und die noch zu verteilende Ressourcenmenge ist auf 5,0 gesunken. Die grafische Darstellung der Ressourcenverteilung und Performanzprofile bzw. Stapelspeicher nach dem zweiten Rekursionsschritt ist in Abbildung 4.3 zu sehen.

Analog läuft der dritte Rekursionsschritt ab. Hier wird der erste Stützpunkt des Performanzprofils (a) gewählt. Im vierten Rekursionsschritt wird nicht wie bisher einer der ersten Stützpunkte eines jeden Performanzprofils gewählt, sondern der vierte des Performanzprofils (c), da die Qualitätssteigerung bis zu diesem besser ist, als die bis zum beispielsweise ersten Stützpunkt dieses Performanzprofils. Tabelle 4.3 zeigt den Stand nach diesem Schritt.

Performanzprofil (a): Gesamtsteigung des Qualitätsverlaufes vom Stützpunkt (0 | 0) aus.

Stützpunkt:	(1 | 0,8)	(5 | 1)
Steigung:	0,8	0,2

Performanzprofil (b): Gesamtsteigung des Qualitätsverlaufes vom Stützpunkt (0 | 0) aus.

Stützpunkt:	(1 | 0)	(1,2 | 0,4)	(4 | 0,6)	(6 | 1)
Steigung:	0	$0,\overline{3}$	0,15	—

Performanzprofil (c): Gesamtsteigung des Qualitätsverlaufes vom Stützpunkt (0 | 0) aus.

Stützpunkt:	(0,5 | 0,1)	(1,0 | 0,24)	(1,5 | 0,5)	(2 | 1)
Steigung:	0,2	0,24	$0,\overline{3}$	0,5

Performanzprofil (d): Gesamtsteigung des Qualitätsverlaufes vom Stützpunkt (0,2 | 0,4) aus.

Stützpunkt:	(**0,5** | **0,7**)	(1,5 | 0,9)	(5 | 1)
Steigung:	**1,0**	$0,\overline{3846153}$	0,125

Tabelle 4.2.: Steigungen der möglichen Schritte bei der Verteilung von noch 5,3 Ressourcen auf die Beispiel-Performanzprofile im zweiten Rekursionsschritt

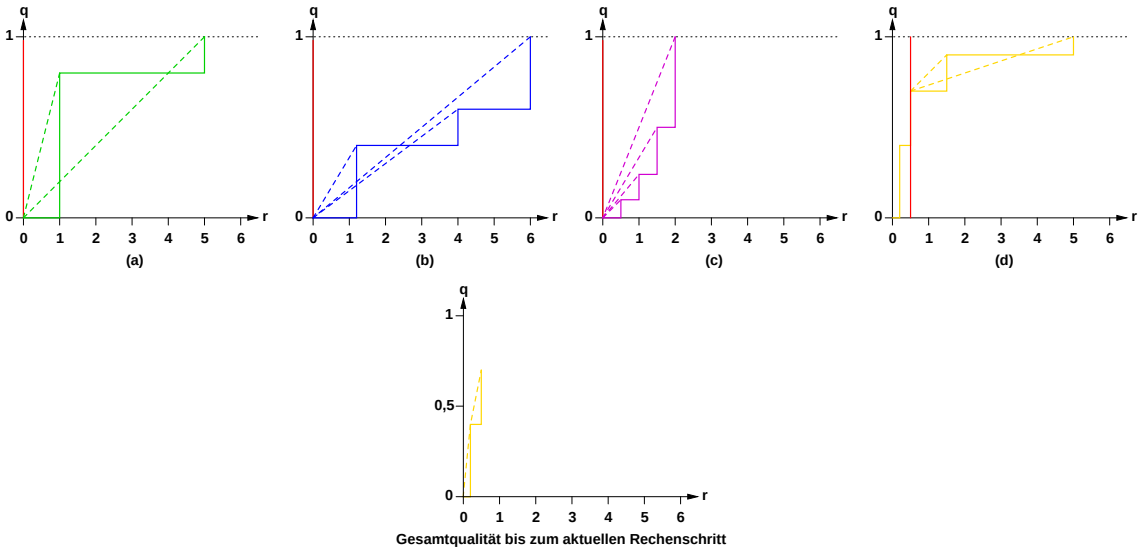

Abbildung 4.3.: Beispiel-Performanzprofile und Zwischenergebnis nach dem zweiten Rekursionsschritt

In der grafischen Darstellung des Endergebnisses in Abbildung 4.4 wird dies hervorgehoben, in dem der gesamte Rekursionsschritt gestrichelt und die einzelnen Teilschritte des gewählten Anytime-Algorithmus normal eingezeichnet sind.

In der Tabelle 4.4 werden sowohl die bisherigen als auch die weiteren Schritte des Rechenbeispiels in kompakter Form dargestellt.

4.3.2.4. Mehrvergabe von Ressourcen

Ein weiterer Vorteil der Treppenstufen-Methode ist, daß relativ einfach berechnet werden kann, ob nicht eine kleine zusätzliche Menge an Ressourcen die Qualität deutlich verbessert: Von der bisher errechneten Verteilung ausgehend wird überprüft, ob jeweils ein weiterer Schritt, also eine weitere Treppenstufe eines der Performanzprofile, die Qualität noch verbessern kann.

Die dafür notwendige Menge an Ressourcen setzt sich aus den eventuell noch übriggebliebenen plus zusätzlichen, gegebenenfalls begrenzten Ressourcen zusammen.

Dazu können verschiedene Ansätze verwendet werden:

- Die Ressourcen werden dem Modul zugesprochen, dessen Performanzprofil im

Performanzprofil (a): Gesamtsteigung des Qualitätsverlaufes vom Stützpunkt $(0 \mid 0)$ aus.

Stützpunkt:	$(5 \mid 1)$
Steigung:	0,05

Performanzprofil (b): Gesamtsteigung des Qualitätsverlaufes vom Stützpunkt $(0 \mid 0)$ aus.

Stützpunkt:	$(1 \mid 0)$	$(1{,}2 \mid 0{,}4)$	$(4 \mid 0{,}6)$	$(6 \mid 1)$
Steigung:	0	$0{,}\overline{3}$	0,15	—

Performanzprofil (c): Gesamtsteigung des Qualitätsverlaufes vom Stützpunkt $(0 \mid 0)$ aus.

Stützpunkt:	$(0{,}5 \mid 0{,}1)$	$(1{,}0 \mid 0{,}24)$	$(1{,}5 \mid 0{,}5)$	$(\mathbf{2} \mid \mathbf{1})$
Steigung:	0,2	0,24	$0{,}\overline{3}$	**0,5**

Performanzprofil (d): Gesamtsteigung des Qualitätsverlaufes vom Stützpunkt $(0{,}2 \mid 0{,}4)$ aus.

Stützpunkt:	$(1{,}5 \mid 0{,}9)$	$(5 \mid 1)$
Steigung:	$0{,}\overline{3846153}$	0,125

Tabelle 4.3.: Steigungen der möglichen Schritte bei der Verteilung von noch 4,0 Ressourcen auf die Beispiel-Performanzprofile im vierten Rekursionsschritt

Nr.	Res.	PP.	St'p.	St.	Res'z.	Q'st.	Ges'q.	(a)	(b)	(c)	(d)
1	5,5	(d)	$(0{,}2 \mid 0{,}4)$	2	0,2	0,4	0,4	0	0	0	0,2
2	5,3	(d)	$(0{,}5 \mid 0{,}7)$	1	0,3	0,3	0,7	0	0	0	0,5
3	5,0	(a)	$(1 \mid 0{,}8)$	0,8	1	0,8	1,5	1	0	0	0,5
4	4,0	(c)	$(2 \mid 1)$	0,5	2	1	2,5	1	0	2	0,5
5	2,0	(b)	$(1{,}2 \mid 0{,}4)$	$0{,}\overline{3}$	1,2	0,4	2,9	1	1,2	2	0,5
6	0,8	*Nicht mehr genügend Ressourcen übrig.*					2,9	1	1,2	2	0,5

Legende siehe Tabelle 4.5.

Tabelle 4.4.: Tabellarischer Verlauf der Ressourcenverteilung

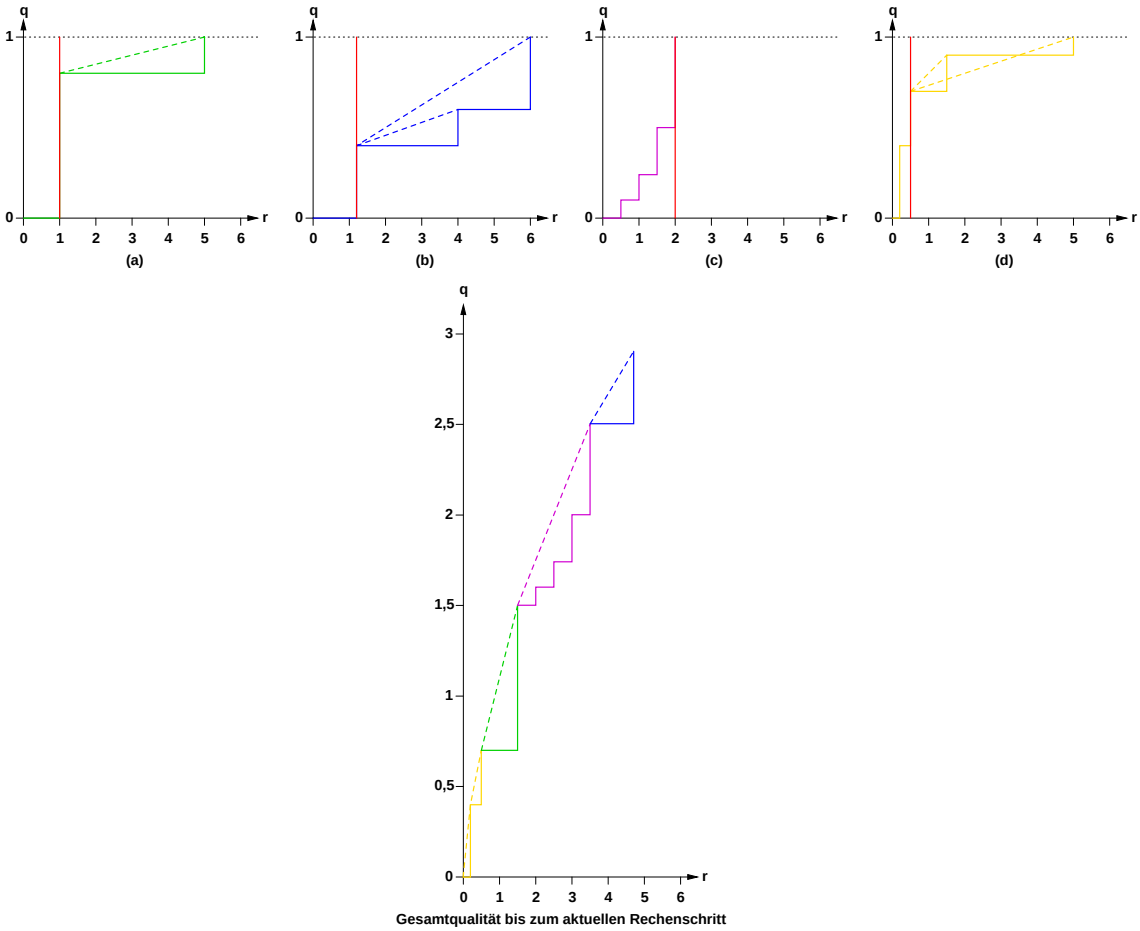

Abbildung 4.4.: Beispiel-Performanzprofile und Ergebnis nach Ende des Algorithmus

Nr.	Nummer des Rekursionsschrittes
Res.	Zu vergebende Ressourcenmenge
PP.	Performanzprofil
St'p.	Gewählter Stützpunkt
St.	Steigung
Res'z.	Menge der dem vorgenannten Performanzprofil in diesem Durchlauf zugeteilten Ressourcen
Q'st.	Qualitätsteigerung gegenüber vorherigem Zwischenergebnis
Ges'q.	Bis zu diesem Rechenschritt erreichte Gesamtqualität des Systems (unnormiert)
(a) bis (d)	Bisher zugewiesene Ressourcenmengen für die Performanzprofile (a) bis (d)

Tabelle 4.5.: Legende zu Tabelle 4.4

nächsten Schritt[7] die größte Qualitätsverbesserung mit sich bringt. Damit werden die zusätzlichen Ressourcen so genutzt, daß abschließend eine möglichst hohe Qualität erreicht wird.

- Die Ressourcen werden dem Modul zugesprochen, dessen Performanzprofil im nächsten Schritt[7] die größte Steigung hat. Dies bedeutet, daß die effizienteste Möglichkeit gewählt wird, um die zusätzlichen Ressourcen noch zu verwenden.
- Die Ressourcen werden dem Modul zugesprochen, dessen Performanzprofil im nächsten Schritt die geringste Menge an Ressourcen verbraucht, also die ursprünglich gesetzte Grenze am wenigsten überschreitet.

Alle drei Ansätze können sowohl mit als auch ohne Beschränkung der zusätzlich zu vergebenden Ressourcen angewendet werden. Allerdings besteht bei der Überschreitung ohne Beschränkung die Gefahr, daß der zusätzliche Schritt sehr viel zusätzliche Ressourcen verteilt[8]. Wird der letzte der drei Ansätze verwendet, besteht außerdem

[7] Sofern dieser Schritt noch innerhalb der noch übriggebliebenen Ressourcen plus einer begrenzten zusätzlichen Menge bleibt.

[8] Beispiel: Seien PP_1 und PP_2 Performanzprofile mit den Stützpunktlisten

$$\$_1 := \{(0 \mid 0); (0{,}05 \mid 0{,}3); (0{,}1 \mid 0{,}4); (0{,}15 \mid 0{,}5); (3 \mid 0{,}52)\}$$

und

$$\$_2 := \{(0 \mid 0); (0{,}1 \mid 0{,}1); (0{,}2 \mid 0{,}2)\}$$

Ist nun die ursprünglich zu verteilende Ressourcenmenge z. B. 0,45 Ressourceneinheiten, dann werden bei diesem Ansatz trotzdem 3 Ressourceneinheiten verteilt, obwohl dies nur noch eine sehr geringe Qualitätssteigerung mit sich bringt. Allgemein können solche Fälle dann auftreten,

die Möglichkeit, daß die dadurch erreichte Qualitätssteigerung nur sehr gering ist und sich der zusätzliche Ressourcenaufwand gegebenenfalls nicht gelohnt hat.

Die zusätzliche Menge an zu vergebenden Ressourcen sollte abhängig von der ursprünglich zu verteilenden Menge an Ressourcen sein, da das Programm die Ressourceneinheit — und damit den Bereich, in dem sich die Ressourcenwerte abspielen — nicht kennt.

Weiterhin kann diese zusätzliche Ressourcenmenge auch noch von der Menge der nach den bisherigen Berechnungen verteilten bzw. übriggeblieben Ressourcen abhängig gemacht werden. Dies kann aber gegebenenfalls dazu führen, daß mehr zu vergebende Ressourcen bei sonst gleichgebliebenen Parametern zu einer schlechteren errechneten Gesamtqualität führen. Grund hierfür ist die Möglichkeit, daß trotz einer größeren, anfänglich zur Verfügung stehenden Ressourcenmenge die insgesamt zu vergebende Ressourcenmenge nach Berechnung der zusätzlich zu vergebenden Ressourcenmenge kleiner ist. Dies hat zur Folge, daß damit erzeugte Qualitätsverläufe nicht notwendigerweise monoton steigend sind und damit nicht mehr die Eigenschaften eines Performanzprofils haben. Eine Lösung dieses auch bei Austauschheuristiken vorkommenden Problems wird in Abschnitt 4.4.2 beschrieben.

In ORCAN V2 kann die Abhängigkeit von den beiden beschriebenen Werten anhand zweier Common-Lisp-Keyword-Parameter festgelegt werden. Die Details der Implementierung obiger Ansätze und der Begrenzung der Ressourcenüberschreitung finden sich im Abschnitt 5.5.

4.3.2.5. Bewertung

Wird ORCAN nicht nur zur Kompilierung von Anytime-Algorithmen in ein Anytime-Modul verwendet, sondern zur Generierung von Ressourcenverteilung in Echtzeitsystemen (z. B. anhand von dynamisch generierten Performanzprofilen), so besteht die Möglichkeit, durch Einsatz von ORCAN als Anytime-Algorithmus sämtliche Zwischenergebnisse der Treppenstufen-Methode zu nutzen. Dies ergibt speziell bei dieser Methode Sinn, da es sich um ein Greedy-Verfahren handelt und somit jedes errechnete Zwischenergebnis bereits ein Teil des endgültigen Ergebnisses ist. Dies wiederum bedeutet, daß jedes Zwischenergebnis eine vorzeitige Zusicherung ist, wieviel Ressourcen bestimmte Anytime-Algorithmen auf jeden Fall bekommen. So kann man diejenigen Anytime-Algorithmen, die bereits in den Zwischenergebnissen viele Ressourcen zugeteilt bekommen, bereits während der Berechnung der Ressourcenverteilung starten, sofern dies die Verknüpfungen der Anytime-Algorithmen untereinander

wenn die Abstände zwischen den Stützpunkten sehr unterschiedlich sind bzw. sich immer mal wieder größere Sprünge der r-Werte in den Stützpunktlisten finden.

zulassen.

Ein weiterer Vorteil ist, daß in den von der Treppenstufen-Methode errechneten Daten bereits eine Empfehlung für die Reihenfolge, in der die Ressourcen von den einzelnen Anytime-Algorithmen verbraucht werden, vorhanden ist: Addiert man die den Anytime-Algorithmen zugeordneten Ressourcen nicht sofort auf, sondern merkt sich jede einzelne Zuordnung und deren Reihenfolge[9] und gibt diese Daten anstatt der üblichen Ressourcenverteilung zurück, so kann man die Ressourcen anfangs von Anytime-Algorithmen verbrauchen lassen, die schnell eine große Qualitätssteigerung mit sich bringen und gegen Ende erst die Anytime-Algorithmen zum Zuge kommen lassen, die nicht mehr sehr viel zur Qualitätssteigerung beitragen. Wird nun die Ausführung des mit diesen Daten kompilierten Anytime-Moduls vorzeitig abgebrochen (weil beispielsweise die zur Verfügung stehenden Ressourcen geringer waren als ursprünglich vermutet), so wurden nur die Anytime-Algorithmen mit der geringsten Qualitätsverbesserung ausgelassen und das vom Anytime-Modul errechnete Ergebnis hat nur wenig an Qualität eingebüßt.

Stehen im Gegensatz dazu bei der Ausführung des Anytime-Moduls mehr Ressourcen zur Verfügung als ursprünglich angenommen, so können durch Verwendung der Mehrvergabe-Funktion des Treppenstufen-Verfahrens die zusätzlichen Ressourcen demjenigen Anytime-Algorithmen zugeordnet werden, der am meisten damit anfangen kann.

4.4. Behandlung von Sonderfällen

4.4.1. Zu geringe zu vergebende Ressourcenmenge

Bei sehr geringen Mengen an zu vergebenden Ressourcen kann es vorkommen, daß diese kleiner sind als der r-Wert des kleinsten Stützpunktes mit $r > 0$.

Für diesen Fall ist keine der bisher beschriebenen Methoden sehr geeignet:

- Die lineare und die exponentielle Regression approximieren das Performanzprofil als Ganzes, was gerade bei der Betrachtung kleiner Ressourcenmengen zu starken Abweichungen vom ursprünglichen Performanzprofil führt.
- Die Hillclimbing-Methode und die abschnittsweise lineare Methode geben allen

[9] Siehe auch Algorithmus 2. Er ist bereits so formuliert, daß das Aufsummieren erst direkt vor der Rückgabe (Zeilen 37 bis 39) geschieht.

Modulen gleichviel (bei `:combine #'*`) oder einem (bzw. mehreren bei gleicher Steigung) Modul alles (bei `:combine #'+`).

- Die Treppenstufen-Methode vergibt keine Ressourcen, da bei der Treppenstufen-Interpretation bei der zu verteilenden Menge an Ressourcen keine der Stufen erreicht werden kann. Dies liegt daran, daß alle Stützpunkte bzw. Stufen erst bei mehr verfügbaren Ressourcen zu erreichen sind.

Entsprechend ist es nicht sinnvoll, diese Methoden mit ihren teils umfangreichen Berechnungen zu benutzen, wenn deren Verhalten bzw. errechnete Verteilung auf einfache Weise voraussagbar ist. Dazu kommt, daß die errechnete Verteilung unbrauchbar sein kann, beispielsweise, falls gar keine Ressourcen verteilt wurden.

Da die vorhandenen Daten im Ressourcenbereich vor dem ersten Stützpunkt nur als sehr ungenau angenommen werden können, kann auch die Komplexität der Berechnung stark reduziert werden. Es bieten sich hierbei die folgenden zwei einfachen Berechnungsverfahren an:

- Eine *korrekte Verteilung* wird in dieser Arbeit eine Ressourcenverteilung genannt, bei der die zu vergebenden Ressourcen gleichmäßig unter den Performanzprofilen mit der größten Steigung aufgeteilt werden. Dies entspricht einer Maximierung der zu erwartenden Qualität.

  Zum Beispiel werden 0,1 Ressourceneinheiten auf vier Performanzprofile, deren erste Stützpunkte mit $r > 0$ bei $(1 \mid 0)$, $(0{,}2 \mid 0{,}2)$, $(0{,}5 \mid 1{,}25)$ und $(2 \mid 5)$ liegen (also die Steigungen 0; 1; 2,5 und 2,5 haben), wie folgt *korrekt* verteilt: Die ersten beiden Performanzprofile bekommen keine Ressourceneinheiten zugewiesen, das dritte und vierte jeweils 0,05 Ressourceneinheiten.

  Wird die Addition für die Verknüpfung der Performanzprofile verwendet, so erreicht diese Art der Verteilung zwar die aufgrund der gegebenen Performanzprofile die größtmögliche Qualität, weist aber den meisten Anytime-Algorithmen gar keine Ressourcen zu.

  Somit ist diese Art der Verteilung nur für Anytime-Module geeignet, bei denen es keine Abhängigkeiten zwischen den einzelnen Anytime-Algorithmen gibt.

- Eine *faire Verteilung* wird in dieser Arbeit eine Ressourcenverteilung genannt, deren Ressourcenwerte proportional zur Steigung zwischen den ersten beiden Stützpunkten des entsprechenden Performanzprofils sind.

  0,1 Ressourceneinheiten werden auf dieselben vier Performanzprofile wie im obengenannten Beispiel wie folgt *fair* verteilt: Das erste Performanzprofil bekommt keine Ressourceneinheiten zugewiesen, das zweite $\frac{2}{120} = 0{,}01\overline{6}$ Ressourceneinheiten, das dritte und vierte je $\frac{5}{120} = 0{,}041\overline{6}$ Ressourceneinheiten.

Diese Art der Verteilung gewährleistet, daß einerseits alle Anytime-Algorithmen Ressourcen zugewiesen bekommen, deren Qualitätssteigerung bis zum ersten Stützpunkt mit $r > 0$ größer als Null ist, und andererseits, daß die Anytime-Algorithmen, die eine höhere Qualitätssteigerung aufweisen, auch mehr Ressourcen bekommen. Dies entspricht der größtmöglichen Qualität, wenn als Verknüpfungsfunktion zwischen den Performanzprofilen die Multiplikation verwendet wird[10].

Da bei dieser Art der Verteilung alle Anytime-Algorithmen, die zur Qualität beitragen, Ressourcen zugeteilt bekommen, ist sie auch für Anytime-Module geeignet, deren Anytime-Algorithmen Abhängigkeiten untereinander aufweisen.

Im allgemeinen ist eine faire Verteilung einer korrekten vorzuziehen, da es in den meisten Fällen Abhängigkeiten zwischen den einzelnen Anytime-Algorithmen eines Anytime-Moduls gibt, also das Ergebnis eines Anytime-Algorithmus als Eingabe für einen anderen verwendet wird. Werden in diesem Fall durch die korrekte Verteilung ausschließlich dem zweiten Algorithmus Ressourcen zugeteilt, so kann dieser trotz der ihm zugeteilten Ressourcen kein verwendbares Ergebnis liefern, da er vom ersten Anytime-Algorithmus keine verwendbaren Eingabewerte bekommen hat.

4.4.2. Nicht monoton steigende Qualitätsverläufe

Bei der Berechnung der in Abschnitt 2.6 bereits erwähnten System-Performanzprofilen kann ein Problem auftreten: Wird zur Kompilierung eine Methode verwendet, deren Qualitätsverläufe nicht notwendigerweise monoton steigend sein müssen — wie beispielsweise bei der Hillclimbing-Methode, welche sich in lokalen Extrema verfangen kann — so muß eine größere zur Verfügung stehende Ressourcenmenge nicht notwendigerweise zu einer höheren oder gleichhohen Qualität führen, da jede Änderung der Parameter — auch die Änderung der zur Verfügung stehenden Ressourcenmenge — zu einem anderen Pfad und damit auch zum Finden eines anderen, gegebenenfalls nur lokalen Extremwertes führen kann.

Dies widerspricht wiederum der Eigenschaft von Performanzprofilen, immer monoton steigend zu sein. Somit sind z. B. mit dem Hillclimbing-Verfahren erzeugte Qualitätsverläufe nicht notwendigerweise Performanzprofile. Ein Beispiel für solch

[10] Dies gilt nur, sofern kein Performanzprofil im Abschnitt zwischen den ersten beiden Stützpunkten eine Steigung von Null hat. Sollte dies aber der Fall sein, so wird bei diesem Verfahren die größtmögliche Qualität erreicht, wenn bei der Multiplikation der Qualitätswerte sämtliche Qualitätswerte gleich Null — d. h. sämtliche Performanzprofile mit Steigung gleich Null im oben genannten Abschnitt — ausgelassen werden.

einen nicht monoton steigenden Qualitätsverlauf findet sich in Abbildung 4.5a[11]. Die oben genannten Artefakte spiegeln sich im Qualitätsverlauf durch ein Absinken des Graphen wider.

Dieses Problem kann bei System-Performanzprofilen in Form von Stützpunktlisten sehr einfach gelöst werden, indem bei der Generierung der Stützpunkte des System-Performanzprofils die Qualität eines berechneten Stützpunktes gemerkt und nach der Berechnung des nächsten überprüft wird, ob dieser eine bessere Qualität aufweist. Ist dies der Fall, so wird er dem System-Performanzprofil hinzugefügt, andernfalls wird er einfach ignoriert und der darauffolgende Stützpunkt berechnet.

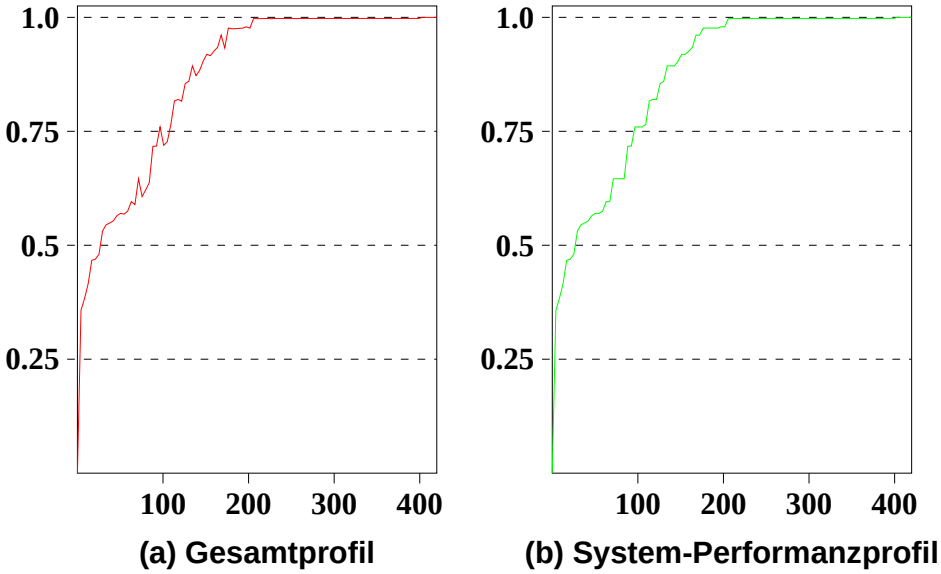

Abbildung 4.5.: Beispiel für einen mit der Hillclimbing-Methode erzeugten Qualitätsverlauf und das daraus resultierende System-Performanzprofil

Bei der Ausführung des Anytime-Moduls mit einer gegebenen Ressourcenmenge sucht es sich den größten Ressourcenwert im System-Performanzprofil, der gleich oder kleiner der zu vergebenden Ressourcenmenge ist. Somit wird garantiert, daß das Anytime-Modul trotz des errechneten nicht monoton steigenden Qualitätsverlaufes ein monoton steigendes System-Performanzprofil hat und damit mit einer größeren zur Verfügung stehenden Ressourcenmenge immer eine gleichhohe oder höhere Ergebnisqualität liefert. In Abbildung 4.5 finden sich ein nicht monoton steigender Qualitätsverlauf und das daraus errechnete, monoton steigende System-Performanzprofil.

[11] Technische Hinweise zum Lesen der von ORCAN V2 erzeugten Diagramme finden sich in Abschnitt 6.1.

5. Implementierung

5.1. Entwicklungsumgebung

ORCAN wurde geschrieben in der Programmiersprache Common Lisp [Steele, 1994] und unter Verwendung von Liquid Common Lisp [LIQUID, 1997] (ehemals Lucid Common Lisp) und Allegro Common Lisp 5.0 [ACL, 1996] entwickelt. Die grafische Ausgabe der Laufzeitmessungen erfolgt im Adobe PostScript® Format [Adobe Systems Inc., 1991a], Version 2.0 und ist damit gut geeignet für die direkte, plattformunabhängige Weiterverwendung der Grafiken in TEX-, PostScript®- und anderen Dokumenten, sowie zum sofortigen Ausdrucken.

Die Möglichkeit von Laufzeit- und Speicherverbrauchmessungen ist nicht Bestandteil von Common Lisp, sondern — falls vorhanden — von der jeweiligen Implementation abhängig. Für die Messungen von Laufzeit und Speicherverbrauch wurden in ORCAN V2 die Funktionen `#'monitor-inclusive-consing` und `#'monitor-inclusive-time` des Monitoring-Tool [HP-Lucid, 1990, Seite 2·24] von Liquid Common Lisp verwendet. Entsprechend funktionieren die von der ORCAN-Funktion `#'rtt` durchgeführten Messungen nur beim Ausführen von ORCAN unter Liquid oder Lucid Common Lisp.

Die Messungen für diese Arbeit fanden hauptsächlich unter Solaris 2.6 auf Sun Sparc Stations 10 und 20 statt. Weitere Testläufe ohne Laufzeitmessung (ausschließlich zum Darstellen der errechneten Ergebnisse) wurden mit Allegro Common Lisp 4.2 und 5.0 unter Solaris 2.7 auf Sun Ultra Sparcs 10 und 20 sowie unter Linux 2.x auf x86-Architekturen durchgeführt.

5.2. Erweiterungen gegenüber Orcan V1

In ORCAN V1 [Baus & Beckert, 1998] sind vier Kompilierungsmethoden — die linear-regressive, die exponentiell-regressive, die Hillclimbing- und die abschnittsweise lineare Kompilierungsmethode — sowie absolute Stützpunktlisten als Eingabeformat implementiert.

ORCAN V2 baut auf ORCAN V1 auf und erweitert dessen Grundfunktionalität um die Treppenstufen-Methode mit ihrer Möglichkeit zur Mehrvergabe von Ressourcen, vier weitere Eingabeformate (Parameter für lineare Funktionen, Parameter für exponentielle Funktionen, Common Lisp Lambda-Ausdrücke und relative Stützpunktlisten) sowie die Behandlung des Falles, daß die zu vergebende Ressourcenmenge zu gering ist, um mit den gegebenen Methoden sinnvoll kompiliert zu werden. Zusätzlich können Performanzprofile von Beginn an von der Kompilierung mit einer bestimmten Ressourcenmenge ausgeschlossen werden, wenn anhand der zu vergebenden Ressourcenmenge festgestellt wurde, daß die gegebene Ressourcenmenge für ihre Ausführung nicht ausreicht. Außerdem kann ORCAN V2 (im Normalfall durch Rechnen an der Grenze der Rechengenauigkeit entstandene) negative zu vergebende Ressourcenmengen verarbeiten.

Weiter gehört zu ORCAN V2 eine Umgebung zum Erstellen von System-Performanzprofilen und zur Durchführung, Speicherung und grafischen Darstellung von Laufzeittests für das Vergleichen verschiedener Kompilierungsmethoden, Parameter und Performanzprofile. Ebenso gehört zu dieser Umgebung ein Zufallsgenerator für Performanzprofile, mit denen beispielsweise Laufzeitmessungen für neue Kompilierungsmethoden durchgeführt werden können.

5.3. Kombinationsmöglichkeiten von Eingabeformaten und Berechnungsmethoden

ORCAN bietet verschiedene Kombinationsmöglichkeiten von Eingabeformaten und Kompilierungsmethoden an. Es gibt folgende fünf Eingabeformate:

`:pp` Performanzprofile in Form von absoluten Stützpunktlisten mit Stützpunkten (Qualität | Ressourcen). Einschränkungen: Die Stützpunkte müssen in der Reihenfolge ihrer r-Werte stehen und die q-Werte müssen monoton steigend sein.

Beispiel für ein Performanzprofil:

Normale Notation: $((0 \mid 0), (0{,}1 \mid 0{,}5), (1 \mid 0{,}6), (3 \mid 0{,}9), (4 \mid 1))$
Lisp-Notation: `'((0 0) (0.1 0.5) (1 0.6) (3 0.9) (4 1))`

`:abs` Performanzprofile in Form von relativen[1] Stützpunktlisten mit dem r-Wert und der Steigung zwischen dem betrachteten und dem darauffolgenden Stützpunkt als Elemente. Der erste Stützpunkt kann auch drei Elemente haben; diese sind dann r-Wert, y-Achsenabschnitt und Steigung. Stützpunktlisten in diesem Format werden zur Weiterverarbeitung innerhalb von ORCAN in absolute Stützpunktlisten umgewandelt und auch im weiteren wie diese behandelt. Der Quelltext der Konvertierungsfunktion `#'alc` findet sich ab Seite 174 in der Quelldatei `orcan-helpers.lisp` (Abschnitt C.10).

Beispiele für Performanzprofile in dieser Form:

Normale Notation: $((0;0{,}1;1{,}3), (0{,}25;0{,}5), (0{,}5;0{,}75))$
$((0;1), (0{,}25;0{,}5), (0{,}5;0{,}75))$
Lisp-Notation: `'((0 0.1 1.3) (0.25 0.5) (0.5 0.75))`
`'((0 1) (0.25 0.5) (0.5 0.75))`

`:lin` Performanzprofile in Form von Parametern für lineare Funktionen (Steigung, q-Achsenabschnitt) der Art $\mathrm{q}(r) = \max(0; \min(1; q_0 + mr))$, wobei q_0 der q-Achsenabschnitt und m die Steigung ist. Einschränkungen: $q_0 \in \mathbb{R}$, $m \in \mathbb{R}_0^+$

Beispiele für Performanzprofile in dieser Form:

Normale Notation: $(0{,}25;0)$
$(1;0{,}5)$
Lisp-Notation: `'(0.25 0)`
`'(1 0.5)`

`:exp` Performanzprofile in Form von Parameter η und λ für Exponential-Funktionen der Art $\mathrm{q}(r) = \max\left(0; \min\left(1; 1 - \eta e^{-\lambda r}\right)\right)$. Einschränkungen: η,$\lambda \in \mathbb{R}_0^+$

Beispiele für Performanzprofile in dieser Form:

Normale Notation: $(1;1)$
$(2;0{,}5)$
$(0{,}5;2)$
Lisp-Notation: `'(1 1)`
`'(2 0.5)`
`'(0.5 2)`

[1] „abs“ steht für „abschnittsweise“.

`:lambda` Performanzprofile in Form von Funktionen bzw. Lambda-Ausdrücken. Einschränkungen: Die durch die Lambda-Ausdrücke gegebenen Funktionen müssen monoton steigend sein.

Beispiel für ein Performanzprofil:

```
#'(lambda (x) (min 0 (max 1 (- 1 (exp -x)))))
```
[2]

Die verschiedenen Kompilierungsmethoden verarbeiten intern jeweils nur eines dieser Eingabeformate:

`:abs` Abschnittsweise lineare Kompilierungsmethode: Stützpunktlisten

`:lin` lineare Kompilierungsmethode: Parameter für lineare Funktionen

`:exp` exponentielle Kompilierungsmethode: Parameter für exponentielle Funktionen

`:hill` Hillclimbing-Methode: Lambda-Ausdrücke

`:stair` Treppenstufen-Methode: Stützpunktlisten

Damit trotzdem die große Anzahl an Kombinationsmöglichkeiten von Eingabeformaten und Kompilierungsmethoden genutzt werden kann, gibt es Konvertierungsverfahren für die obengenannten Eingabeformate. Folgende Konvertierungen sind verlustfrei:

Relative Stützpunktlisten können direkt in äquivalente absolute Stützpunktlisten konvertiert werden und umgekehrt. Da aber ORCAN ausschließlich auf absoluten Stützpunktlisten arbeitet, ist die Umkehrrichtung nicht implementiert.

Lineare Funktionsparameter können in Stützpunktlisten (mit maximal drei Stützpunkten[3]) konvertiert werden. Dies ergibt allerdings nur dann Sinn, wenn die Stützpunktlisten danach auch als abschnittsweise linear interpretiert werden. Aus demselben Grund wie oben ist nur die Konvertierung in absolute Stützpunktlisten implementiert.

[2] Die Einschränkung des Funktionsergebnisses auf das Intervall $[0;1]$ wird von ORCAN vorgenommen, ist hier aber nochmals zur Verdeutlichung angegeben.

[3] Die drei Stützpunkte sind die Punkte mit $q = 0$, $q = 1$ und $r = 0$. Sofern die Punkte mit $q = 0$ und $q = 1$ nicht existieren, reicht theoretisch der Punkt mit $r = 0$ als Performanzprofil aus. Außerdem kann es vorkommen, daß der Punkt mit $r = 0$ mit einem der beiden anderen zusammenfällt.

Alle Eingabeformate können in Lambda-Ausdrücke konvertiert werden: Funktionsparameter werden in die entsprechenden Funktionen eingesetzt und Stützpunktlisten in abschnittsweise lineare Funktionen umgewandelt.

Lambda-Ausdrücke können nicht konvertiert werden, d. h. sie dienen ausschließlich der Hillclimbing-Methode als Eingabeformat.

Verlustbehaftet sind die beiden Regressionsverfahren zur Umwandlung von Stützpunktlisten in Funktionsparameter:

Zu den Stützpunktlisten können durch Regression jeweils die Parameter der linearen bzw. exponentiellen Funktion gefunden werden, die die geringste Abweichung gegenüber dem als Stützpunktliste gegebenen Performanzprofil hat [Baus & Beckert, 1998].

Exponentielle Funktionsparameter können in Stützpunktlisten konvertiert werden, indem für eine Reihe von r-Werten die entsprechenden q-Werte ausgerechnet werden. Entsprechend gilt: Je mehr Stützpunkte ausgerechnet werden, desto geringer wird die Abweichung von der durch die exponentiellen Funktionsparameter gegebenen Funktion. Auch hier ergibt die Konvertierung nur dann Sinn, wenn die Stützpunktlisten danach auch als abschnittsweise linear interpretiert werden.

Da sich die Abweichungen bei zwei verlustbehafteten Konvertierungen sehr schnell hochschaukeln können, wurde auf die Konvertierung von Parameter exponentieller Funktionen in Parameter linearer Funktionen verzichtet. Auf die umgekehrte Richtung wurde ebenfalls verzichtet, da es auf jeden Fall zu größeren Abweichungen vom Original-Profil kommt.

Weiter wurde auf die Verwendung von Funktionsparametern bei der abschnittsweise linearen Kompilierungsmethode verzichtet, da die Verwendung von Parametern linearer Funktionen – bis auf die Konvertierung der Daten – der direkten Verwendung der Parameter in der linearen Methode entspricht.

Auch hat die Verwendung der als Funktionsparameter gegebenen Performanzprofile im Treppenstufen-Verfahren wenig Sinn, da bei der Konvertierung dieser Parameter in Stützpunktlisten die minimal notwendige Anzahl von Stützpunkten verwendet wird, was die Treppenstufen-Methode wiederum als einen einzigen Schritt, in dem die gesamte Qualität erreicht wird, interpretiert. Entsprechend würde sie aus Daten in solch einem Eingabeformat keinerlei brauchbare Ergebnisse errechnen.

Eine Übersicht über den Datenfluß in ORCAN V2 findet sich in Abbildung 5.1.

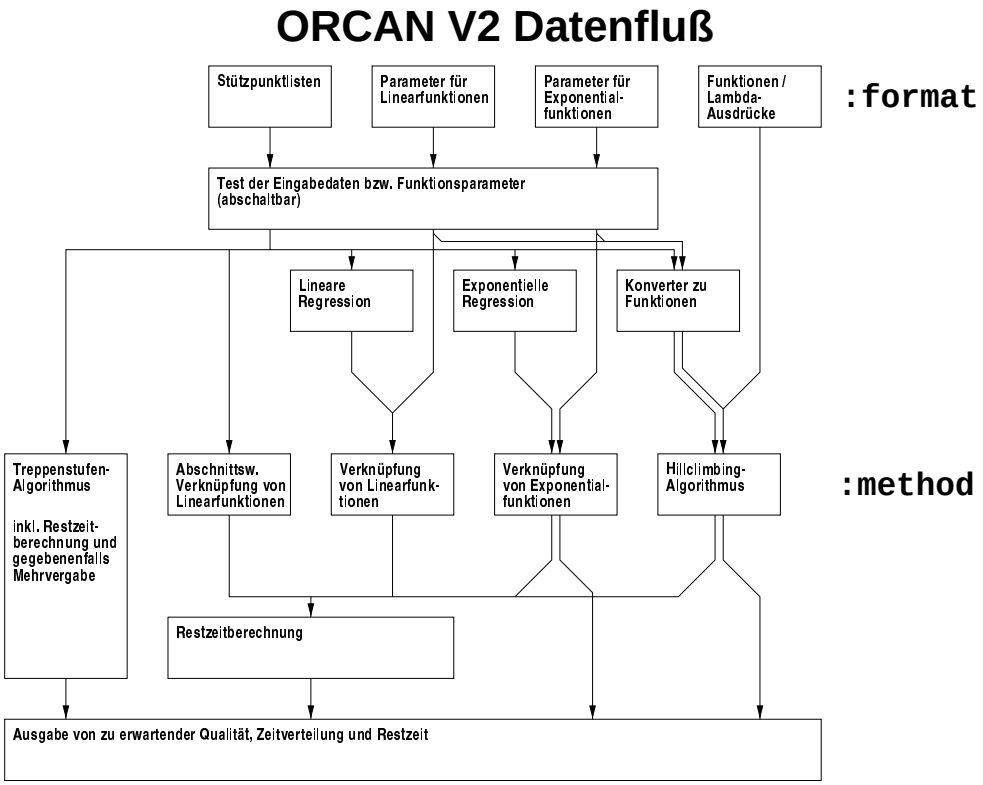

Abbildung 5.1.: Datenfluß in ORCAN V2

5.4. Sonderfall zu wenig zu vergebende Ressourcen

Ist die Menge der zu verteilenden Ressourcen kleiner ist als der r-Wert des kleinsten Stützpunktes mit $r > 0$, so berechnet ORCAN V2 die Ressourcenverteilung nicht durch die normalen Kompilierungsmethoden, sondern — wie in Abschnitt 4.4.1 beschrieben — durch ein wesentlich einfacheres Verfahren. Ob dabei eine *faire* oder *korrekte* Verteilung vorgenommen wird, wird durch den Keyword-Parameter `:distribute-on-small-interval` festgelegt: Er kann einen der beiden Werte `:fair` oder `:correct` übergeben bekommen. Die zu erwartende Qualität wird dann durch die über den Keyword-Parameter `:combine` angegebene Funktion berechnet.

Da die Kombination von `:distribute-on-small-interval :correct` und `:combine #'*` zu unbrauchbaren Ergebnissen führen kann, wird in diesem Fall eine Warnung ausgegeben. Sie kann auch per Keyword-Parameter abgeschaltet werden. Siehe auch Abschnitt B Interface.

5.5. Interpretation von Stützpunktlisten als Treppenstufen

Die Implementation der Treppenstufen-Methode entspricht dem in Abschnitt 4.3.2 beschriebenen Verfahren. In den folgenden beiden Abschnitten werden noch Details der Implementation der Mehrvergabe von Ressourcen beschrieben.

Die Mehrvergabe geschieht in ORCAN V2 durch die Funktion `#'last-step`[4]. Sie wird auf jeden Fall aufgerufen, sobald das Treppenstufen-Verfahren keine weiteren Stufen innerhalb der noch zu vergebenden Zeit findet, auch wenn keine Mehrvergabe verwendet wird.

5.5.1. Implementierung der verschiedenen Ansätze zur Mehrvergabe

Alle in Abschnitt 4.3.2.4 beschriebenen Ansätze zur Mehrvergabe von Ressourcen bei der Treppenstufen-Methode sind in ORCAN V2 implementiert. Ob und — falls ja — nach welchem dieser Ansätze ORCAN zusätzliche Ressourcenmengen verteilt werden, wird durch den Keyword-Parameter `:type` bestimmt:

[4] Quelltext auf Seite 151

`:none` Es wird keine zusätzliche Ressourcenmenge verteilt. *[Defaulteinstellung]*

`:mint` Es wird die kleinstmögliche Überschreitung[5] verwendet.

`:abmt` Es wird die kleinstmögliche Überschreitung[6] verwendet, sofern eine Überschreitung innerhalb des gegebenen Intervalls möglich ist.

`:abqu` Es wird eine zusätzliche Ressourcenmenge[7] — falls innerhalb des gegebenen Intervalls — dem Modul gegeben, das in einem Schritt die größte Qualitätssteigerung bringt.

`:abgr` Es wird eine zusätzliche Ressourcenmenge[8] — falls innerhalb des gegebenen Intervalls — dem Modul gegeben, das in einem Schritt die effizienteste Qualitätssteigerung bringt.

Wird ein anderer Wert als `:none` verwendet, sind negative Restressourcenwerte möglich. Sie geben an, wieviele Ressourcen zusätzlich vergeben wurden.

5.5.2. Berechnung der zulässigen Überschreitung bei der Mehrvergabe

Die zusätzliche Menge an Ressourcen sollte abhängig sein von der ursprünglich zu verteilenden Menge an Ressourcen res_{bisher} und eventuell auch von der Menge der nach den bisherigen Berechnungen übriggeblieben Ressourcen $res_{\text{übrig bisher}} = res_{\text{bisher}} - res_{\text{verbraucht}}$, da das Programm die Ressourceneinheit — und damit den Bereich, in dem sich die Ressourcenwerte abspielen — nicht kennt.

In ORCAN wird die erlaubte Überschreitung der ursprünglich zu verteilenden Ressourcenmenge res_{bisher} um $res_{\text{zusätzlich}}$ und damit auch die noch zu verteilende Ressourcenmenge inklusive der Überschreitung $res_{\text{übrig neu}} := res_{\text{bisher}} + res_{\text{zusätzlich}}$ durch die zwei Keyword-Parameter `:alpha` und `:beta` festgelegt:

$$res_{\text{zusätzlich}} := (\alpha \cdot res_{\text{bisher}}) + (\beta \cdot res_{\text{übrig bisher}}) \tag{5.1}$$

$$\begin{aligned} res_{\text{übrig neu}} &:= res_{\text{übrig bisher}} + (\alpha \cdot res_{\text{bisher}}) + (\beta \cdot res_{\text{übrig bisher}}) \\ &= (\alpha \cdot res_{\text{bisher}}) + ((\beta + 1) \cdot res_{\text{übrig bisher}}) \end{aligned} \tag{5.2}$$

[5] `:mint` steht für „minimal time“.
[6] `:abmt` steht für „alpha beta“ und „minimal time“.
[7] `:abqu` steht für „alpha beta“ und „quality“.
[8] `:abgr` steht für „alpha beta“ und „gradient“.

Ein Wert von $\beta > 0$ ist sinnvoll, falls eine hohe Überschreitung der zu vergebenden Ressourcenmenge nur in dem Fall erwünscht ist, daß ohne Mehrvergabe viele Ressourcen unverbraucht blieben. Dagegen sorgt ein Wert von $\alpha > 0$ dafür, daß die mögliche Überschreitung sich proportional zu den zu vergebenden Ressourcen verhält, d. h. bei wenigen zur Verfügung stehenden Ressourcen ist auch die mögliche Überschreitung klein, und bei vielen zur Verfügung stehenden Ressourcen ist die Überschreitung entsprechend groß.

Beispiele für die Verwendung von α und β finden sich in Tabelle 5.1.

α	β	Entspricht
0	0	den Keyword-Parametern `:type :none`
1	0	dem neuen Gesamtintervall $[0; 2res_{\text{bisher}}]$
0	1	dem neuen Gesamtintervall $[0; res_{\text{bisher}} + res_{\text{übrig bisher}}]$

Tabelle 5.1.: Beispiele für die Verwendung von α und β

5.6. Eingabeformate

Bei der Implementation der zusätzlichen Eingabeformate in ORCAN V2 trat das Problem auf, daß die beiden regressiven Kompilierungsmethoden von ORCAN V1 das Gesamtprofil zweier Performanzprofile nicht als Parameter η und λ bzw. Steigung und q-Achsenabschnitt, sondern in Form einer Stützpunktliste weitergeben. Als r-Werte für deren Stützpunkte verwendet ORCAN V1 die Vereinigung der Mengen aller x-Werte der einzelnen Profile.

Diese Werte sind aber bei der Übergabe der Performanzprofile in Form von Funktionsparametern nicht vorhanden, so daß in diesem Fall die Werte auf einem anderen Weg generiert werden müssen.

Im folgenden wird gezeigt, daß zwei beliebige r-Werte hierfür ausreichen.

Im Fall der linear-regressiven Kompilierungsmethode ist dies trivial, da jede Gerade durch zwei auf ihr liegende Punkte eindeutig definiert ist.

Dies gilt ebenso für die exponentiell-regressive Kompilierungsmethode, wie sich aus den in Abschnitt 2.5.2.2 verwendeten Formeln leicht zeigen läßt. Nach Formel 2.4f. kann jede Funktion der Art $\mathrm{q}(r) = 1 - \eta e^{-\lambda r}$ in eine lineare Funktion $\bar{q} = \bar{\eta} + \bar{\lambda} r$ mit $\bar{q} = \ln(1 - q)$, $\bar{\eta} = \ln(\eta)$ und $\bar{\lambda} = -\lambda$ umgeformt werden.

Zur Berechnung der Werte $\bar{\eta}$ und $\bar{\lambda}$ reicht es aus, wenn zwei Wertepaare $(\, \bar{q}_1 \mid r_1 \,)$ und

$(\bar{q}_2 \mid r_2)$ dieser Funktion bekannt sind. $\bar{q}$ kann aus q, η aus $\bar{\eta}$ und λ aus $\bar{\lambda}$ berechnet werden.

Somit reichen zwei beliebige Wertepaare $(q_1 \mid r_1)$ und $(q_2 \mid r_2)$ einer Funktion $\mathrm{q}(r) = 1 - \eta e^{-\lambda r}$ aus, um deren Parameter η und λ zu bestimmen.

In ORCAN V2 sind für die exponentiell-regressive Kompilierungsmethode die r-Werte 0 und 1 als Default-Ergebnisse vorgegeben, da die q-Werte für diese r-Werte für die meisten Performanzprofile im Bereich zwischen 0 und 1 liegen. Sollten sich unter den Performanzprofilen extreme Profile befinden, bei denen der q-Wert für einen dieser r-Werte außerhalb des Wertebereiches der verwendeten Common-Lisp-Implementation liegt, so kann über den Parameter `:expx` eine andere Liste von x-Werten verwendet werden.

Bei der linear-regressiven Kompilierungsmethode wählt ORCAN V2 die r-Werte so, daß es sich bei den beiden q-Werten um für die Funktion markante Werte[9] handelt.

5.7. Berechnung der für Anytime-Module notwendigen Daten

Neben der Funktion `#'distribute` zum Berechnen einzelner Ressourcenverteilungen besitzt ORCAN V2 auch eine Funktion namens `#'spp`[10] zum Berechnen der für Anytime-Module notwendigen System-Performanzprofile. Die System-Performanzprofile werden berechnet, indem innerhalb eines berechneten oder gegebenen Ressourcenintervalls für eine Reihe von Ressourcenmengen Ressourcenverteilungen berechnet und gespeichert werden.

Sind alle notwendigen Ressourcenverteilungen berechnet, so gibt `#'spp` eine Liste aus Paaren mit Ressourcenwerten und den dazugehörigen Ergebnissen der Funktion `#'distribute` — bestehend aus zu erwartender Qualität, Ressourcenverteilung und Restressourcen — zurück, welche später bei der Ausführung des entsprechenden Anytime-Moduls verwendet wird, um die Ressourcen auf die Subkomponenten zu verteilen.

[9] Es handelt sich hierbei um die Schnittpunkte mit den Geraden $q = 1$ und $q = 0$. Sollte letzterer einen negativen r-Wert haben, wird stattdessen $r = 0$ verwendet.

[10] `spp` steht für „System Performance Profile“.

5.8. Umgebung zur Durchführung von Laufzeittests

Ähnlich der im letzten Abschnitt beschriebenen Funktion `#'spp` arbeitet die Funktion `#'rtt`[11], welche ganze Serien von Laufzeitmessungen durchführt und zum Auswerten neuer Kompilierungsmethoden oder Eingabeformate verwendet werden kann. Im Unterschied zur Funktion `#'spp` berechnet `#'rtt` jedoch nicht nur die entsprechende Ressourcenverteilung, sondern berechnet sie für eine Menge unterschiedlicher Parameterwerte und mißt mit dem Liquid Common Lisp Monitoring-Tool auch noch die Dauer der Berechnung sowie deren Speicherverbrauch[12].

Weiter kann `#'rtt` die Laufzeittests wahlweise auf gegebenen oder dynamisch erstellten, zufälligen Performanzprofilen durchführen. Die Generierung von zufälligen Performanzprofile in Form von Stützpunktlisten geschieht durch die zufällige Wahl der Abstände der Stützpunkte in beiden Dimensionen (Ressourcen und Qualität) sowie durch das Wählen der Anzahl der Stützpunkte eines Performanzprofils nach Zufallskriterien. Bei den Performanzprofil-Formaten mit Funktionsparametern werden diese direkt durch einen Zufallsgenerator bestimmt.

Anstatt abschließend die errechnete Ressourcenverteilung zurückzugeben, speichert `#'rtt` die Eingabedaten, die berechneten Ressourcenverteilungen und die Meßwerte in einer Datei ab. Sämtliche darin gespeicherten Daten können später mit der Funktion `#'ps-page` grafisch ausgegeben werden: Neben Qualität, Laufzeit und Speicherverbrauch werden auch die Restressourcen, die verwendeten Profile[13] und die errechnete Ressourcenverteilung in Diagrammen dargestellt.

Weitere Funktionen ermöglichen die fließbandartige Erstellung von Laufzeitdiagrammen mit einem einzigen Funktionsaufruf durch die automatische Generierung von zufälligen Performanzprofilen, Durchführung von Laufzeittests und Erstellung der Grafiken. Sämtliche in dieser Arbeit gezeigten Laufzeitdiagramme sind auf diese Weise erstellt worden.

[11] `rtt` steht für „Run Time Test“.

[12] Da die Messung der Laufzeit und Anzahl der verwendeten `conses` — also des Speicherverbrauches — ausschließlich mit dem Monitoring-Tool [HP-Lucid, 1990, Seite 2·21ff.] von Liquid Common Lisp bzw. dessen Vorgänger Lucid Common Lisp funktionieren, werden diese Messungen unter anderen Lisp-Derivaten automatisch deaktiviert.

[13] Entsprechend der Interpretation der verwendeten Methode werden die Performanzprofile abschnittsweise linear oder als Treppenstufen dargestellt.

5.9. Messung sehr kurzer Laufzeiten

Bei der Durchführung der Laufzeittests kamen die Meßwerte — insbesondere, wenn die Ressourcen nur auf wenige Anytime-Algorithmen verteilt werden sollten — in Bereiche, die deutlich unterhalb der Meßgenauigkeit[14] des Liquid Common Lisp Monitoring-Tool lagen.

Dieses Problem kann umgangen werden, indem die gleiche Berechnung mehrmals[15] hintereinander ausgeführt wird und die über die gesamte Dauer gemessene Laufzeit durch die Anzahl der Durchläufe geteilt wird. Da aber auch Messungen mit sehr lange dauernden Berechnungen durchgeführt wurden, mußte die mehrfache Ausführung von Berechnungen selektiv geschehen. In ORCAN V2 wird die Anzahl der Mehrfachausführungen abhängig von der Anzahl der verwendeten Performanzprofile gewählt.

Trotz dieser Maßnahme lagen die Laufzeitwerte der Treppenstufen-Methode häufig in einem eng an die Meßgenauigkeit grenzenden Bereich, so daß für die Laufzeitmessung des Treppenstufen-Verfahrens die Anzahl der Mehrfachausführung um einen weiteren Faktor erhöht werden mußte.

5.10. Sonstige Erweiterungen

5.10.1. Akzeptanz von negativen r-Werten in Performanzprofilen

Mit dem neuen Keyword-Parameter `:accept-negative-time` der Funktion `#'distribute` kann festgelegt werden, ob negative Werte des Parameters ***time***[16] als 0 betrachtet werden (Keyword-Parameter `:accept-negative-time T`) oder ob – wie bisher – das Programm mit einer Fehlermeldung abbricht (Keyword-Parameter `:accept-negative-time nil`, *Defaulteinstellung*).

Dies ist insbesondere dann nützlich, wenn bei der Berechnung der zu vergebenden Ressourcenmenge am Rande der Rechengenauigkeit der verwendeten Common-Lisp-Implementation gearbeitet wurde und so gegebenenfalls die zu vergebende Ressour-

[14] Die kleinste mit dem Liquid Common Lisp Monitoring-Tool meßbare Zeiteinheit liegt bei 0,1 Sekunden.

[15] Anzahl der Wiederholungen durch Parameter bestimmbar. Siehe Seite 145.

[16] ORCAN wurde — wie bereits erwähnt — vor dem Hintergrund entwickelt, daß die zu verteilende Ressource Rechenzeit ist. Aus diesem Grund taucht der Begriff *„time“* in vielen Variablen- und Parameternamen auf.

cenmenge minimal unterhalb von Null liegen können.

5.10.2. Ausschluß von Performanzprofilen von der Berechnung von Ressourcenverteilungen

Bei Performanzprofilen in Form von Stützpunktlisten kann relativ leicht festgestellt werden, ob sie innerhalb der gegebenen Ressourcenmenge überhaupt eine Qualität größer Null erreichen. Erreichen sie diese nicht, so können Rechenressourcen gespart werden, indem die entsprechenden Performanzprofile im voraus von der Kompilierung ausgeschlossen werden, da sie nicht zur Qualitätssteigerung beitragen können.

Dieser Ausschluß ist allerdings nur dann möglich, wenn die Qualitätsfunktion beliebig viele Parameter akzeptiert und diese gleichbehandelt, wie es z. B. die Funktionen `#'+` und `#'*` tun. Und selbst dann ist zu beachten, daß die Ergebnisse mit und ohne diese Performanzprofile unterschiedlich sein können. Als Beispiel sei hier die Funktion `#'*` genannt, die ohne den Ausschluß dieser Performanzprofile nur eine Qualität von Null liefert, mit Ausschluß aber eine Qualität größer Null. Aus diesem Grund ist diese Funktionalität von ORCAN V2 nur mit Vorsicht zu verwenden.

Gesteuert wird sie über den neuen Keyword-Parameter `:test-if-necessary`. Mit ihm kann festgelegt werden, ob Performanzprofile schon im voraus von der Berechnung ausgeschlossen werden (Keyword-Parameter `:test-if-necessary T`, *Defaulteinstellung*), wenn sie auch mit der kompletten zu vergebenden Ressourcenmenge nur eine Qualität von Null erbringen. Andernfalls werden alle Performanzprofile zur Kompilierung zugelassen, und die oben beschriebene Überprüfung findet nicht statt (Keyword-Parameter `:test-if-necessary nil`).

5.11. Anwendung

Das ursprüngliche Anwendungsgebiet von ORCAN ist, in einem Kompiler nach Zilberstein (1993) die von dem zu kompilierenden Anytime-Modul benötigten Ressourcenverteilungen zu berechnen. Da ORCAN selbst als Anytime-Algorithmus fungieren kann, ist es ebenfalls möglich, ORCAN in dynamischen Szenarien zu verwenden, in denen eine Kompilierung von Anytime-Algorithmen in Echtzeit notwendig ist.

5.11.1. Anwendungsbeispiel

Angewendet wird ORCAN in „JAMES“[17] [Wittig, 1998], welches die Werkzeuge zur Entwicklung beliebiger Anytime-Systeme bereitstellt. JAMES generiert Prozesse für die einzelnen Teilaufgaben dieser Systeme und weist ihnen Ressourcen anhand einer von ORCAN errechneten Ressourcenverteilung zu, die sie während ihrer Berechnung nutzen können. Da permanent neue Daten über das Verhalten der einzelnen Subkomponenten des Anytime-Systems in die verwendeten Performanzprofile einfließen, werden auch die Ressourcenverteilungen dynamisch berechnet. Von Vorteil ist dabei, daß ORCAN anytime-fähig ist und so ebenfalls als Subkomponente eines Anytime-Systems fungieren kann.

Mit JAMES konstruiert wurde beispielsweise das System BOLA[18] [Blocher, 1999], welches zur natürlichsprachlichen Beantwortung von *Wo*-Fragen eines Benutzers bezüglich eines Objektes dient. BOLA baut auf dem Objektlokalisationssystem OLS [Gapp, 1997] auf und erweitert es u. a. um Anytime-Eigenschaften, die es zu einem Echtzeitsystem zur Generierung von Ortsbeschreibungen machen.

5.11.2. Weitere Einsatzmöglichkeiten

Neben der Kompilierung von Anytime-Algorithmen kann ORCAN auch genauso für andere Zwecke eingesetzt werden, bei denen Ressourcen anhand von Performanzprofilen auf verschiedene Ressourcenkonsumenten möglichst optimal verteilt werden müssen.

Beispielsweise könnte die Bandbreite der Internet-Anbindung einer Firma so auf verschiedene Internet-Dienste (E-Mail, WWW, FTP, Telnet, etc.) verteilt werden, daß es zu keiner Zeit zu Leistungseinbußen kommt, weil wegen der intensiven Nutzung der Bandbreite durch weniger wichtigere Dienste (z. B. Herunterladen von Musik- oder Videodaten aus dem WWW) wesentlich wichtigere Dienste (z. B. E-Mail, Rechnerfernsteuerung per Telnet) zum Erliegen kommen.

Weitere Beispiele wären die Aufteilung von Stromressourcen auf die einzelnen Komponenten eines autonomen Roboters oder die Verteilung von begrenztem Speicherplatz auf verschiedene Wissensquellen in einem KI-System mit einer heterogenen Wissensbasis.

[17] Java Anytime Management & Editor System
[18] Beschränkt-optimaler Lokalisationsagent

6. Analyse der verschiedenen Kompilierungsmethoden

6.1. Technische Hinweise zum Lesen der Beispieldiagramme

Die grafische Darstellung eines mit ORCAN durchgeführten Laufzeittests ist in mehrere kleine, in Zeilen und Spalten angeordnete Einzeldiagramme (Schwarze, zyanfarbene und gelbe Rahmen in Abbildung 6.1) unterteilt.

Dabei hat die abgesetzte, erste Spalte eine gesonderte Stellung: Als erstes zeigt sie Informationen, die allen in der Abbildung gezeigten Testdurchläufen gemein sind, wie zum Beispiel die für diesen Laufzeittest verwendeten Performanzprofile (Zyanfarbener Rahmen in Abbildung 6.1). Werden in einem Test verschiedene Kompilierungsmethoden miteinander verglichen, so werden die Performanzprofile in den verschiedenen Interpretationen der Methoden dargestellt. Werden dagegen die verschiedenen Parameter der einzelnen Kompilierungsmethoden miteinander verglichen, so werden die Performanzprofile in der Interpretation der verwendeten Methode dargestellt. In dieser Arbeit werden Stützpunktlisten zwischen den Stützpunkten nur linear oder treppenstufenförmig interpretiert[1], so daß es maximal zwei Diagramme mit den verschiedenen Darstellungen der verwendeten Performanzprofile gibt.

Darunter befindet sich eine Zusammenfassung, in der die Qualitätsverläufe der gezeigten Testdurchläufe übereinander gelegt wurden (Gelber Rahmen in Abbildung 6.1). So können die Unterschiede genauer festgestellt werden. Die Farben in diesem Diagramm entsprechen den in der zweiten Spalte (siehe unten) verwendeten Farben.

Abschließend beherbergt die erste Spalte noch Informationen über Zeitpunkt, Da-

[1] Die linear-regressive und die exponentiell-regressive Kompilierungsmethode verwenden die einzelnen Stützpunkte als Ausgangsdaten für die Regression. Es ist also für sie keine spezielle Interpretationsweise im Bereich zwischen zwei aufeinanderfolgenden Stützpunkten notwendig.

teiname, iterierten Parameter und die verschiedenen verwendeten Werten des Laufzeittests (Roter Rahmen in Abbildung 6.1).

Der zweite zusammenhängende Block von Diagrammen (Schwarze bzw. blaue und grüne Rahmen in Abbildung 6.1) zeigt pro Zeile die Daten zu einem Testdurchlauf (Blaue Rahmen in Abbildung 6.1). Welchen Wert der iterierte Parameter für die jeweiligen Testdurchläufe angenommen hat, ist zwischen der ersten und zweiten Spalte der Abbildung angegeben. Auf der Querachse hat jedes Einzeldiagramm Ressourceneinheiten (RU steht für *„Resource Unit“*) aufgetragen.

Jede dieser Zeilen mit Einzeldiagrammen wurde erzeugt, indem für die angegebene Variante in einem gegebenen Intervall für 100 Ressourcenwerte in gleichem Abstand einzelne Laufzeitmessungen durchgeführt wurden. Ist als Intervall z. B. $[0; 198]$ gegeben, so wurden Messungen mit 0, 2, 4, 6, ..., 198 zu vergebenden Ressourceneinheiten gemacht. Das heißt, daß die ORCAN-Funktion `#'distribute` mit der für die in der entsprechenden Zeile dargestellte Parameter-Variante nacheinander mit jedem dieser Ressourcenwerte als Parameter *time* aufgerufen und die errechneten bzw. gemessenen Werte in das Diagramm auf der Hochachse eingetragen wurden.

Die einzelnen Diagramme jeder Zeile stellen jeweils die im folgenden aufgeführten Werte im Verhältnis zu den zu vergebenden Ressourceneinheiten dar. Dabei stehen Diagramme gleicher Art jeweils in der gleichen Spalte des Blockes[2] (Grüne Rahmen in Abbildung 6.1). Auch die Maßstäbe von Diagrammen gleicher Art sind innerhalb einer Abbildung gleich, jedoch von Abbildung zu Abbildung unterschiedlich[3]. Im folgenden werden die einzelnen Diagramm-Arten näher erläutert:

2. Spalte — Zu erwartende Qualität: Das erste Diagramm eines Testdurchlaufes zeigt die zu erwartende Qualität an, die die jeweilige Variante für die in Spalte 1 gezeigten Performanzprofile mit der auf der Querachse aufgetragenen Ressourcenmenge errechnet hat.

Beim Vergleich der Diagramme in dieser Spalte bzw. im Diagramm mit allen Qualitätsverläufe ist zu beachten, daß vor allem bei Verwendung der linearen und der exponentiellen Regression aufgrund der Approximation der Performanzprofile durch eine Funktion der entsprechenden Familie eine zu erwartende Qualität berechnet werden kann, die höher liegt als die maximal erreichbare Qualität. Weiter ist zu bemerken, daß die Verknüpfungsfunktion nicht bei allen Kompilierungsmethoden dieselbe ist, was trotz gleicher oder ähnlicher

[2] Die Numerierung bezieht sich auf alle Spalten des Laufzeittests, zum Hauptblock der Abbildung (Schwarze bzw. blaue und grüne Rahmen in Abbildung 6.1) gehören davon die Spalten 2 bis 5 (Grüne Rahmen in Abbildung 6.1). Entsprechend werden auch nur diese hier aufgelistet.

[3] Ausnahme: Der Maßstab der Qualität ist — bezogen auf die Höhe der Diagramme — immer gleich.

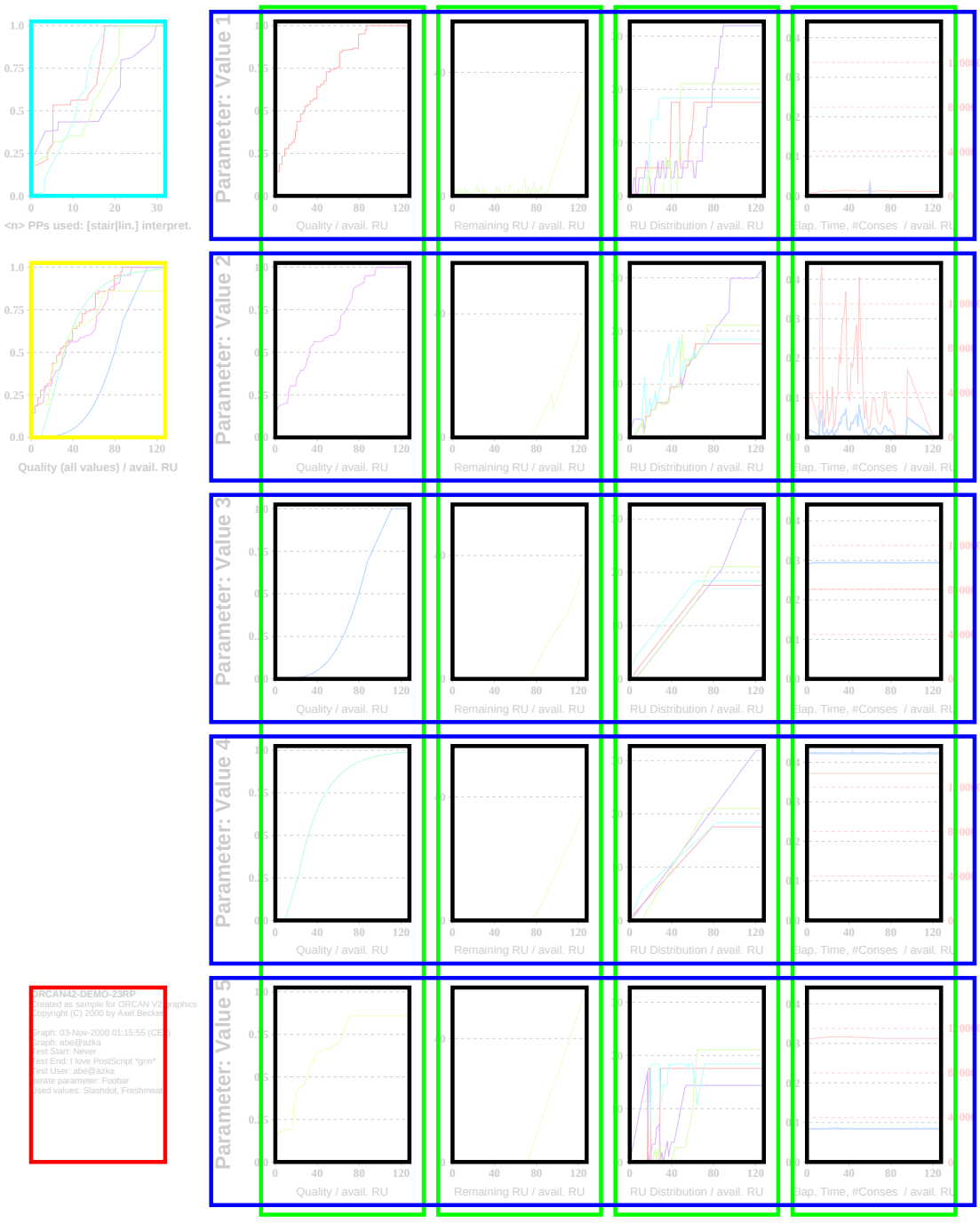

Abbildung 6.1.: Beispiel der grafischen Ausgabe eines ORCAN Laufzeittests

Ressourcenverteilungen zu sehr unterschiedlichen, mal eher konvexen und mal eher konkaven Qualitätsgraphen führen kann.

3. Spalte — Restressourcen: In dieser Spalte werden die bei der errechneten Verteilung übrig gebliebenen Ressourcen angezeigt.

Wird eine Kompilierungsmethode mit Mehrvergabe von Ressourcen verwendet, so können in diesen Diagrammen negative Werte auf der Hochachse aufgetragen werden.

4. Spalte — Ressourcenverteilung: Hier wird die errechnete Verteilung der vergebenen Ressourcen angezeigt. Die errechnete Zuordnung von Ressourcen für jedes in der ersten Spalte angezeigte Performanzprofil wird in diesem Diagramm durch einen Graph in derselben Farbe dargestellt.

5. Spalte — Laufzeit und Speicherverbrauch: Die Diagramme der letzten Spalte zeigen mehrere Graphen unterschiedlicher Art:

In Rot wird der Speicherverbrauch für die Berechnung einer Verteilung entsprechender Ressourcenmengen angezeigt. Die Einheitenskala dafür ist auf der rechten Seite des Diagrammes und hat als Einheit `conses`, welche sich proportional zum Speicherverbrauch verhält [HP-Lucid, 1990, Seite 2·24].

In Blau sind die Laufzeiten von acht nicht direkt nacheinander durchgeführten[4] Laufzeitmessungen eingetragen. Die Einheitenskala hierfür ist auf der linken Seite des Diagrammes und hat als Einheiten Sekunden [HP-Lucid, 1990, Seite 2·24].

Weiter ist anzumerken, daß die Einzeldiagramme, in denen die zu erwartende Qualität (Gelber Rahmen sowie erster grüner Rahmen von links in Abbildung 6.1) aufgeführt ist, System-Performanzprofile sind, deren Generierung aus den Qualitätsverläufen in Abschnitt 4.4.2 näher beschrieben wird.

Bei den dargestellten Performanzprofilen (Zyanfarbener Rahmen in Abbildung 6.1) ist zu beachten, daß sich in diesen Diagrammen dargestellte Qualitätssprünge nicht notwendigerweise bei der gleichen Ressourcenmenge in den Diagrammen mit der zu erwartenden Qualität und der errechneten Ressourcenverteilung widerspiegeln[5]. Dies ist nur dann der Fall, wenn dem jeweiligen Performanzprofil bis zu diesem Punkt sämtliche Ressourcen zugeordnet werden.

[4] Durch die zeitlichen Abstände zwischen den einzelnen Ausführungen der Tests werden die durch andere Prozesse des Systems hervorgerufenen Artefakte eindeutig als solche identifizierbar. Weiter wird dadurch die statistische Signifikanz der Testwerte erhöht.

[5] Erfahrungsgemäß führt dies immer wieder zur Verwirrung des Betrachters.

In Textstellen erwähnte Merkmale der Diagramme sind in denselben teils durch farbige Kreise markiert. Diese Kreise wurden zur Hervorhebung bestimmter Stellen manuell in die Abbildungen eingefügt und sind nicht Bestandteil der grafischen Ausgabe von ORCAN.

Die für die Laufzeittests verwendeten Performanzprofile wurden alle mit dem ORCAN V2 Zufallsgenerator erzeugt. Hierbei wurden bewußt auch Performanzprofile verwendet, deren maximaler q-Wert kleiner als 1 ist, da es diesbezüglich keine Einschränkung in ORCAN gibt.

Die im Text und in den Abbildungsbeschriftungen erwähnten Parameternamen beziehen sich auf die Parameter der ORCAN-Funktion `#'distribute`.

6.2. Allgemeine Feststellungen

Bis auf die Hillclimbing-Methode sind alle[6] Kompilierungsmethoden nahezu unabhängig von der Menge der zu verteilenden Ressourcen. Nur bei sehr kleinen[7] zu verteilenden Ressourcenmengen ist die Laufzeit ungefähr proportional zur verteilten Ressourcenmenge. Danach pendelt sie bei jeder der vier Kompilierungsmethode um einen bestimmten Wert.

Bei der Hillclimbing-Methode dagegen zeigen die Laufzeitdiagramme auffällige Berge und Täler, d. h. die Laufzeit hängt fast gar nicht von der zu vergebenden Ressourcenmenge ab. Dies ist verständlich, da der Hillclimbing-Algorithmus in seinen Entscheidungen, z. B. die Schrittweite herauf oder herab zu setzen, und damit auch seine Laufzeit sehr von den als Parameter übergebenen Performanzprofilen abhängt.

Im Vergleich zwischen Treppenstufen- und Hillclimbing-Verfahren fällt bei vielen Versuchsläufen[8] auf, daß das Treppenstufen-Verfahren große Qualitätssprünge in einem Performanzprofil schon bei wesentlich weniger zu vergebenden Ressourcen erkennt und honoriert (siehe blauer Kreis in Abbildung A.2).

Bei den Laufzeittests mit wenigen Performanzprofilen läßt sich anhand der Sägezahnkurven der Restressourcen gut erkennen, wie das Treppenstufen-Verfahren mit der Verteilung von Ressourcen wartet, bis wieder ein weiterer Stützpunkt im Bereich der zu verteilenden Ressourcen „erscheint“. Nach Vergrößerung des entsprechenden

[6] Die Methode zur Verteilung bei sehr kleinen zur Verfügung stehenden Ressourcenmengen ist bei dieser Betrachtung nicht einbezogen, da bereits ihre Verwendung von der zu verteilenden Ressourcenmenge abhängig ist.

[7] im Vergleich zum größten, in den Stützpunktlisten vorkommenden r-Wert

[8] z. B. bei den Abbildungen A.2ff. (Seiten 109ff.)

Ausschnitt läßt sich dies auch bei den Laufzeittests mit vielen Performanzprofilen erkennen.

In manchen Laufzeittests verdoppelt sich der Ressourcenverbrauch der Treppenstufen-Methode plötzlich und halbiert sich häufig kurz darauf auch wieder. Dies passiert, falls während eines Rekursionsschrittes mehrere Stützpunkte dieselbe Steigung haben und der aktuelle noch zur Verfügung stehende Ressourcenrahmen nur für die Verteilung auf einen der beiden Stützpunkte reicht. Details hierzu sind auf Seite 43 näher beschrieben.

An manchen Stellen gibt es zu Beginn des Qualitätsverlaufes einen Sprung nach oben mit einem darauffolgenden Plateau. Dies entsteht, wenn zu Beginn der Sonderfall eintrat, daß die zu vergebende Ressourcenmenge zu gering war und das in Abschnitt 4.4.1 beschriebene Verfahren eingesetzt wurde, um einen Qualitätswert zu berechnen. Darauffolgende Berechnungen mit der normal gewählten Kompilierungsmethode ergaben anfangs schlechtere Qualitätswerte, wodurch der zweite beschriebene Sonderfall — ein nicht monoton steigender Qualitätsverlauf — eintrat und wie in Abschnitt 4.4.2 beschrieben bearbeitet wurde. In Abbildung A.2 (Seite 109) ist dies bei den beiden regressiven Kompilierungsmethoden erkennbar, bei der linear-regressiven etwas deutlicher.

6.3. Analyse der Methoden aus Ressourcensicht

6.3.1. Laufzeit und Speicherverbrauch allgemein

In bezug auf Laufzeit und Speicherverbrauch fallen sofort zwei Dinge auf: Die Treppenstufen-Methode braucht im Vergleich zu den anderen Kompilierungsmethoden extrem wenig Rechenressourcen, und bei der Hillclimbing-Methode schwanken Laufzeit und Speicherverbrauch während der Berechnung eines einzelnen System-Performanzprofils sehr stark. Die anderen Methoden dagegen brauchen schon kurz nach Beginn der Berechnung für jede Ressourcenverteilung etwa gleichviel Zeit und Speicher.

Der geringe Ressourcenverteilung der Treppenstufen-Methode läßt sich darauf zurückführen, daß es sich bei dieser um einen Greedy-Algorithmus handelt, welcher in einer sehr direkten Weise die Verteilung berechnet. Die abschnittsweise lineare und die beiden regressiven Kompilierungsmethode brauchen viel Rechenressourcen, da sie zuerst rekursiv für jeweils zwei Performanzprofile ein Gesamtprofil erstellen und danach die zu vergebenden Ressourcen rekursiv auf jeweils zwei Gesamtprofile bzw. Performanzprofile verteilen müssen.

Das starke Schwanken der Rechenressourcen der Hillclimbing-Methode hängt damit zusammen, daß sich der „Pfad“, den sich der Hillclimbing-Algorithmus in der Menge der möglichen Ergebnisse sucht, mit jeder Parameteränderung (also auch bei einer Änderung der zur Verfügung stehenden Ressourcenmenge) stark ändern kann.

All diese Beobachtungen lassen sich im Abbildung 6.2 leicht nachvollziehen, aber auch die Abbildungen A.1, A.3 und A.4 (Seiten 108ff.) sowie 6.4 unterstützen diese Aussagen.

6.3.2. Verhältnis von Speicherverbrauch und Laufzeit zueinander

Allgemein zeigt sich in allen Diagrammen, daß sich Laufzeit und Speicherverbrauch bei der Berechnung eines System-Performanzprofils ungefähr proportional verhalten.

Es stellt sich aber genauso heraus, daß die beiden regressiven Kompilierungsmethoden bei etwa gleichem Speicherverbrauch häufig zwei- bis dreimal soviel Rechenzeit wie die abschnittsweise lineare Methode brauchen. Dies zeigt sich deutlich in den Abbildungen 6.3 und A.5 (Seite 112).

Bei der Hillclimbing-Methode dagegen schwankt das Verhältnis zwischen Laufzeit und Speicherverbrauch innerhalb der Berechnung eines System-Performanzprofils zum Teil stärker, wie sich in Abbildung A.10 in der Zeile für `:combine #'+`) erkennen läßt: Im Bereich von 20 bis 40 Ressourceneinheiten klaffen die Diagramme von Laufzeit und Speicherverbrauch wesentlich stärker auseinander als z. B. bei ca. 200 Ressourceneinheiten. Laufzeittestübergreifend bleibt das Verhältnis aber im gleichen Bereich. Beim Vergleich der Abbildungen 6.3 und A.5 läßt sich dies erkennen. Allerdings ist beim Vergleichen zu beachten, daß die Verhältnisse der Laufzeit- und Speicherverbrauchskalen der beiden Diagramme aufgrund unterschiedlicher Spitzenwerte in der jeweiligen Skalenkategorie nicht gleich sind.

Die Treppenstufen-Methode wirft beim Versuch, ihr Verhältnis zwischen Laufzeit- und Speicherverbrauch zu erkennen, ein kleines Problem auf: Die Meßwerte spielen sich in Bereichen ab, in denen sie nur mit einer entsprechenden Vergrößerung der Vektorgrafik abgelesen werden können. Hier können aber auch die Laufzeittests zu Rate gezogen werden, in denen verschiedene Parameterwerte der Treppenstufen-Methode miteinander verglichen werden, da deren Skalen an die Wertebereiche der Treppenstufen-Methode angepaßt sind. So läßt sich beispielsweise in Abbildung A.24 (Seite 131) erkennen, daß das Verhältnis zwischen Laufzeit und Speicherverbrauch im selben Bereich liegt, wie das Verhältnis der Spitzenwerte von Laufzeit und Speicherverbrauch bei der Hillclimbing-Methode, so daß sich selbst bei ausschließlicher

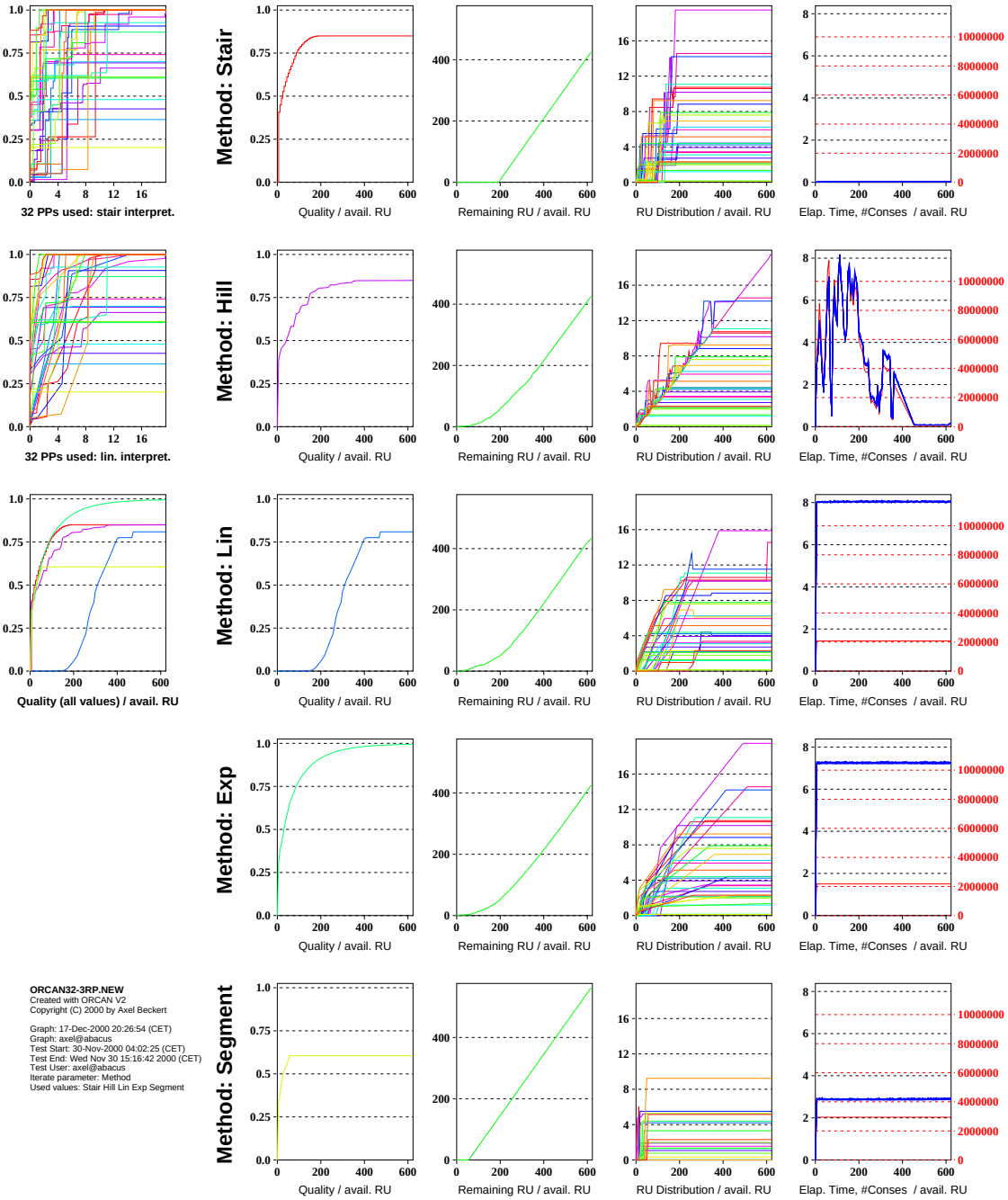

Abbildung 6.2.: Laufzeittest mit 32 Performanzprofilen und Iteration über alle Kompilierungsmethoden

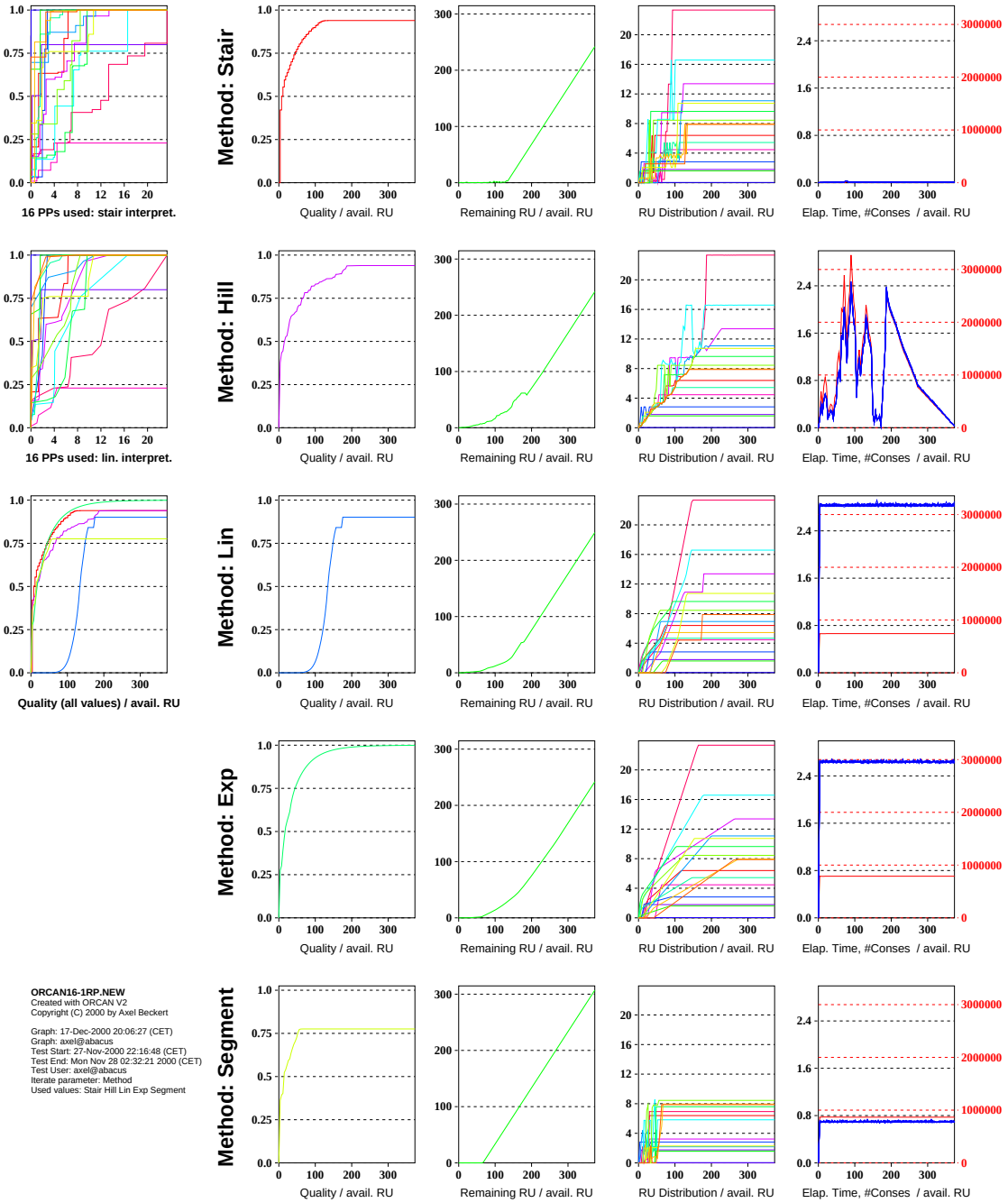

Abbildung 6.3.: Laufzeittest mit 16 Performanzprofilen und Iteration über alle Kompilierungsmethoden

Beschränkung von Rechendauer oder Speicherplatz kein diesbezüglicher Vorteil der Hillclimbing-Methode gegenüber dem Treppenstufen-Verfahren ergibt.

6.3.3. Abhängigkeit von der zu vergebenden Ressourcenmenge

Bei im Vergleich zu den r-Werten der Performanzprofile kleinen zu vergebenden Ressourcenmengen fällt neben bereits in anderen Abschnitten erwähnten Besonderheiten auf, daß die linear-regressive Kompilierungsmethode sowie die Hillclimbing-Methode mit Keyword-Parameter `:combine #'*` aufgrund der Multiplikation als Verknüpfungsfunktion häufig eine zu erwartende Qualität von Null errechnen und deswegen keine Daten zur Berechnung einer sinnvollen Ressourcenverteilung vorliegen haben. Besonders gut erkennbar ist dies in den Abbildungen 6.4 und A.10 (Seite 117).

Das Treppenstufen-Verfahren tendiert allerdings bei wenig zu vergebenden Ressourcen dazu, vielen Anytime-Algorithmen gar keine Ressourcen zuzuteilen, da es einige wenige gefunden hat, die anfangs eine sehr hohe Qualitätssteigerung mit sich bringen. In Abbildung A.4 (Seite 111) läßt sich deutlich erkennen, daß das grüne und gelbe Performanzprofil zu Beginn immer wieder keine Ressourcen zugeteilt bekamen.

Bei den im Vergleich zu den r-Werten der Performanzprofile relativ großen zu vergebenden Ressourcenmengen fällt nur auf, daß der Hillclimbing-Algorithmus im Ressourcenverbrauch kaum noch Ausreißer nach oben hat, was sich dadurch erklären läßt, daß bereits in der Startverteilung die Ressourcen bei fast allen Anytime-Algorithmen für das Erreichen ihrer maximalen Qualität ausreichen und eine Veränderung der Ressourcenverteilung eher zu schlechteren als zu besseren Ergebnissen führt. Dies ist in Abbildung 6.4 ab einer zu vergebenden Ressourcenmenge von ca. 600 Ressourceneinheiten (siehe grüner Kreis) deutlich erkennbar.

6.4. Analyse der Methoden in Abhängigkeit der verwendeten Performanzprofile

6.4.1. Abhängigkeit von der Anzahl der verwendeten Performanzprofile

Wie die Abbildungen A.1 und 6.4 im Vergleich zeigen, so ist bei wenigen Performanzprofilen die Laufzeit, aber auch der Speicherverbrauch bei der exponentiell-

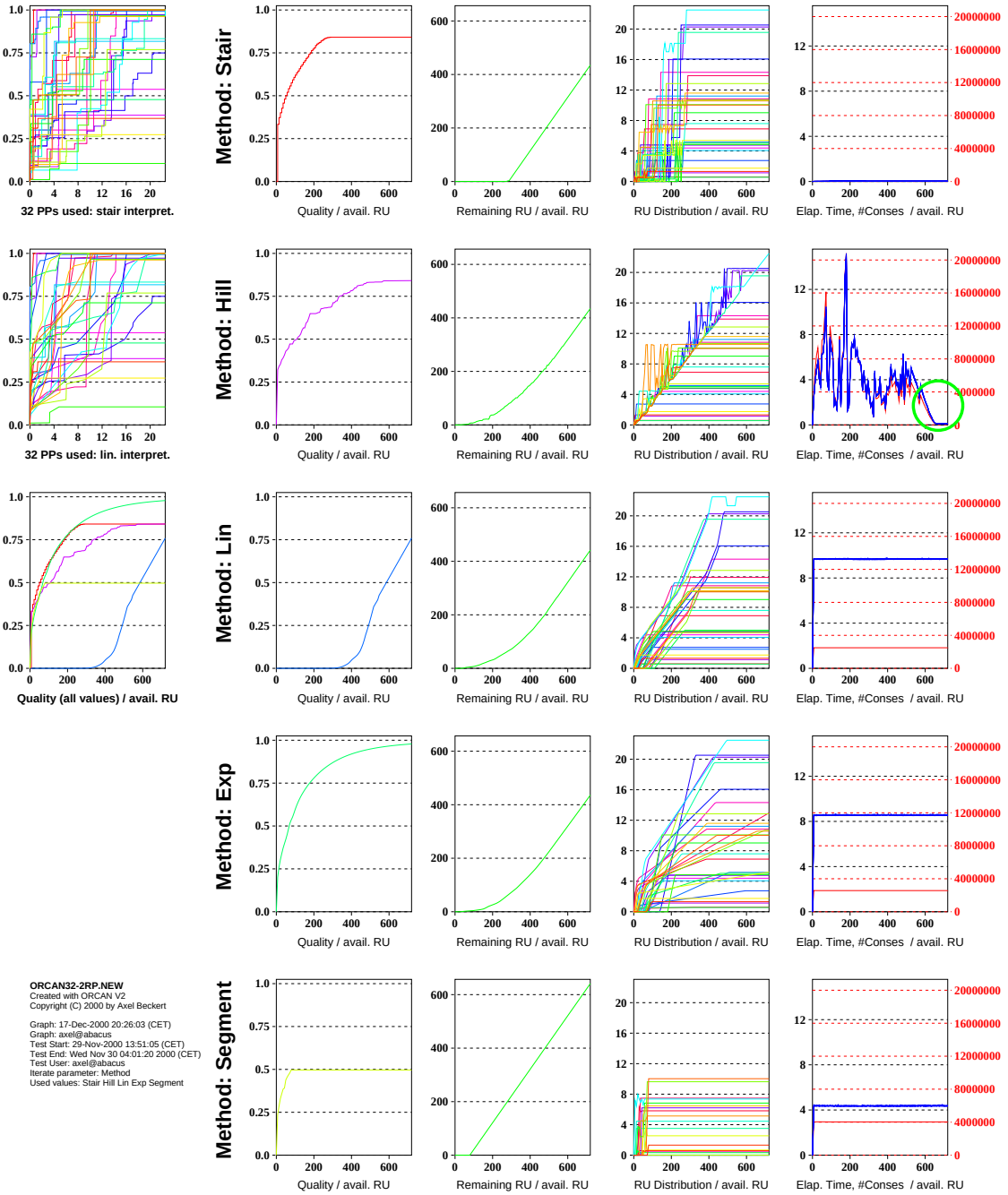

Abbildung 6.4.: Laufzeittest mit 32 Performanzprofilen und Iteration über alle Kompilierungsmethoden

regressiven Kompilierungsmethode gegenüber den anderem Methoden sehr hoch. Bei höheren Anzahlen von Performanzprofilen liegen Laufzeit und Speicherverbrauch der beiden regressiven und der abschnittsweise linearen Kompilierungsmethode etwa gleichauf.

Dies läßt sich darauf zurückführen, daß der Rechenaufwand zum Auf- und Abbauen des Binärbaumes[9] (trotz seiner linearen Abhängigkeit von der Anzahl der Performanzprofile) bei vielen Performanzprofilen im Vergleich zum Berechnen der eigentlichen Ressourcenverteilung zwischen zwei Performanzprofilen überwiegt. Dies ist auch der Grund dafür, daß die Laufzeit und der Speicherverbrauch der abschnittsweise linearen Kompilierungsmethode bei wenigen Performanzprofilen auf dem Niveau des Treppenstufen-Verfahrens liegt, später aber wesentlich höher, jedoch immer noch unter dem Ressourcenverbrauch der beiden regressiven Methoden.

Die Treppenstufen-Methode ist zwar ebenfalls nur linear von der Anzahl der Performanzprofile abhängig, braucht aber insgesamt wesentlich weniger Rechenaufwand und liefert trotzdem durchgängig bessere, weil genauere Ergebnisse.

Bezüglich der errechneten Ressourcenverteilungen läßt sich folgendes bemerken: Während die Hillclimbing-Methode dazu tendiert, bei vielen Performanzprofilen weniger stark von der Ausgangsverteilung abzuweichen, teilt die abschnittsweise lineare Methode bei vielen Performanzprofilen allen relativ wenig Ressourcen zu, was sich bei höheren zu vergebenden Ressourcenmengen auch in der zu erwartenden Qualität negativ niederschlägt. Dies läßt sich darauf zurückführen, daß sich nicht-optimale Verteilungen durch die Aufteilung der Ressourcen über einen Binärbaum verstärken. Bei den beiden regressiven Methoden kommt dies dagegen weniger stark zum Vorschein, da durch deren Approximation der Performanzprofile sich andere „Fehler“ einschleichen, die die Verstärkung von nicht-optimalen Verteilungen bezüglich der zu erwartenden Qualität wieder wettmachen.

6.4.2. Abhängigkeit von der durchschnittlichen Anzahl der Stützpunkte

Bezüglich der durchschnittlichen Anzahl der Stützpunkte läßt sich bemerken, daß insbesondere die abschnittsweise lineare Kompilierungsmethode bei vielen Stützpunkten eine extrem lange Rechendauer hat (siehe Abbildungen 6.5, A.7 und A.8), was sich darauf zurückführen läßt, daß sie für die Berechnung der neuen r-Werte bei der Kombination zweier Performanzprofile die Vereinigungsmenge beider Stütz-

[9] Aufbau durch Berechnen der Gesamtprofile als Knoten eines Binärbaumes, Abbau durch Verteilen der Ressourcen eines jeden Knotens auf seine beiden Äste.

punktlisten verwendet. So kann sich die Anzahl der Stützstellen bei der Kombination zweier Performanzprofile zu einem Gesamtprofil verdoppeln.

Die Laufzeiten und der Speicherverbrauch der Treppenstufen-Methode sind gegenüber der Ressourcenverteilung mit wenigen Stützpunkten zwar merkbar angestiegen, was darauf zurückzuführen ist, daß der Rechenaufwand der Treppenstufen-Methode maximal quadratisch[10] von der Anzahl der Stützstellen abhängt. Bei 128 Stützpunkten pro Performanzprofil lag der Rechenaufwand jedoch nach wie vor eindeutig unter den Werten der beiden regressiven Kompilierungsmethoden.

Die Hillclimbing-Methode hat in etwa den gleichen Ressourcenverbrauch wie die Treppenstufen-Methode, jedoch unterscheiden sich die von der Hillclimbing-Methode errechneten Ressourcenverteilungen nur marginal von den Startwerten. Die von der Treppenstufen-Methode errechneten Ressourcenverteilungen sind dagegen wesentlich differenzierter als die von den anderen Methoden errechneten Verteilungen. Die abschnittsweise lineare Kompilierungsmethode liefert zwar ebenfalls ein differenzierteres Ergebnis als die beiden regressiven und die Hillclimbing-Methode, jedoch liegen — wie oben beschrieben — sowohl Laufzeit als auch Speicherverbrauch in nicht akzeptablen Bereichen.

Bei wenigen Stützpunkten fällt die abschnittsweise lineare Kompilierungsmethode dagegen positiv auf: Sie hat eine niedrige Laufzeit, die wesentlich näher an der der Treppenstufen-Methode als an denen der beiden regressiven Kompilierungsmethoden liegt (siehe Abbildung 6.6). Dies läßt sich darauf zurückführen, daß sich die obengenannte Verdopplung der Stützstellen pro Gesamtprofil bei geringen Stützpunktanzahlen wesentlich geringer auswirkt, als bei hohen. Die Diagramme der anderen Kompilierungsmethoden unterscheiden sich nicht wesentlich von Diagrammen aus anderen Laufzeittests: Die Hillclimbing-Methode fällt durch ihren stark unterschiedlichen Ressourcenverbrauch auf, die Treppenstufen-Methode hat den besten Ressourcenverbrauch bei gleichzeitig differenzierter Ressourcenverteilung, und die beiden regressiven Kompilierungsmethoden liegen beim Ressourcenverbrauch auf den beiden letzten Plätzen.

6.4.3. Verwendung heterogener Performanzprofile

Will man Performanzprofile unterschiedlicher Formate verwenden, so fällt die Hillclimbing-Methode als die vielfältigste Kompilierungsmethode auf, da sie Performanzprofile in Form von Lambda-Ausdrücken verwendet und somit mit jedem Perfor-

[10] Pro Rekursionsschritt vergleicht die Treppenstufen-Methode die Steigungen von maximal $\sum_{i=1}^{|\mathbb{P}|} (|\$_i| - 1)$ Stützpunkten (alle außer den ersten in jedem Performanzprofil), und insgesamt gibt es maximal soviele Rekursionsschritte, wie Stützpunkte vorhanden sind.

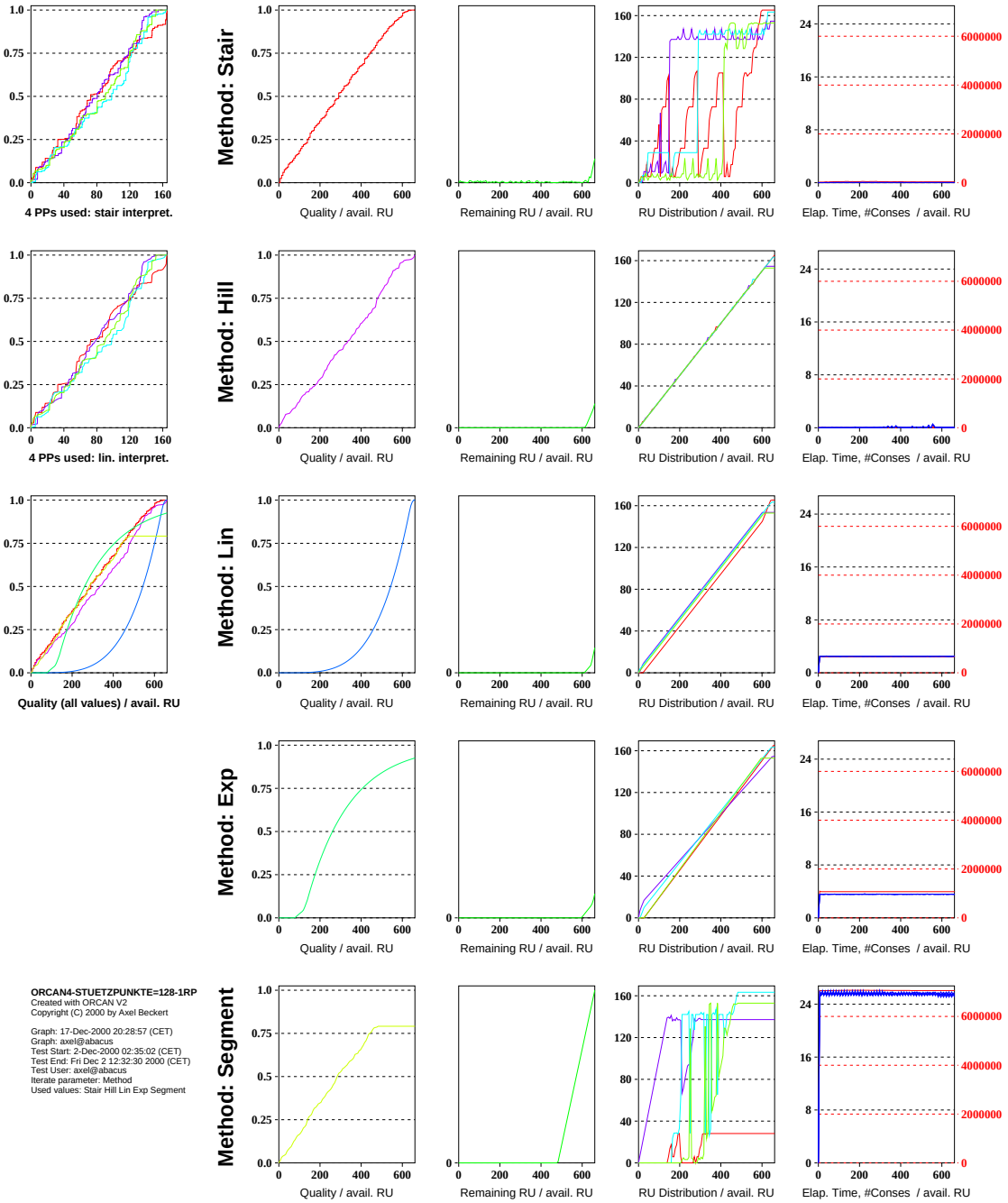

Abbildung 6.5.: Laufzeittest mit 4 Performanzprofilen mit je 128 Stützpunkten und Iteration über alle Kompilierungsmethoden

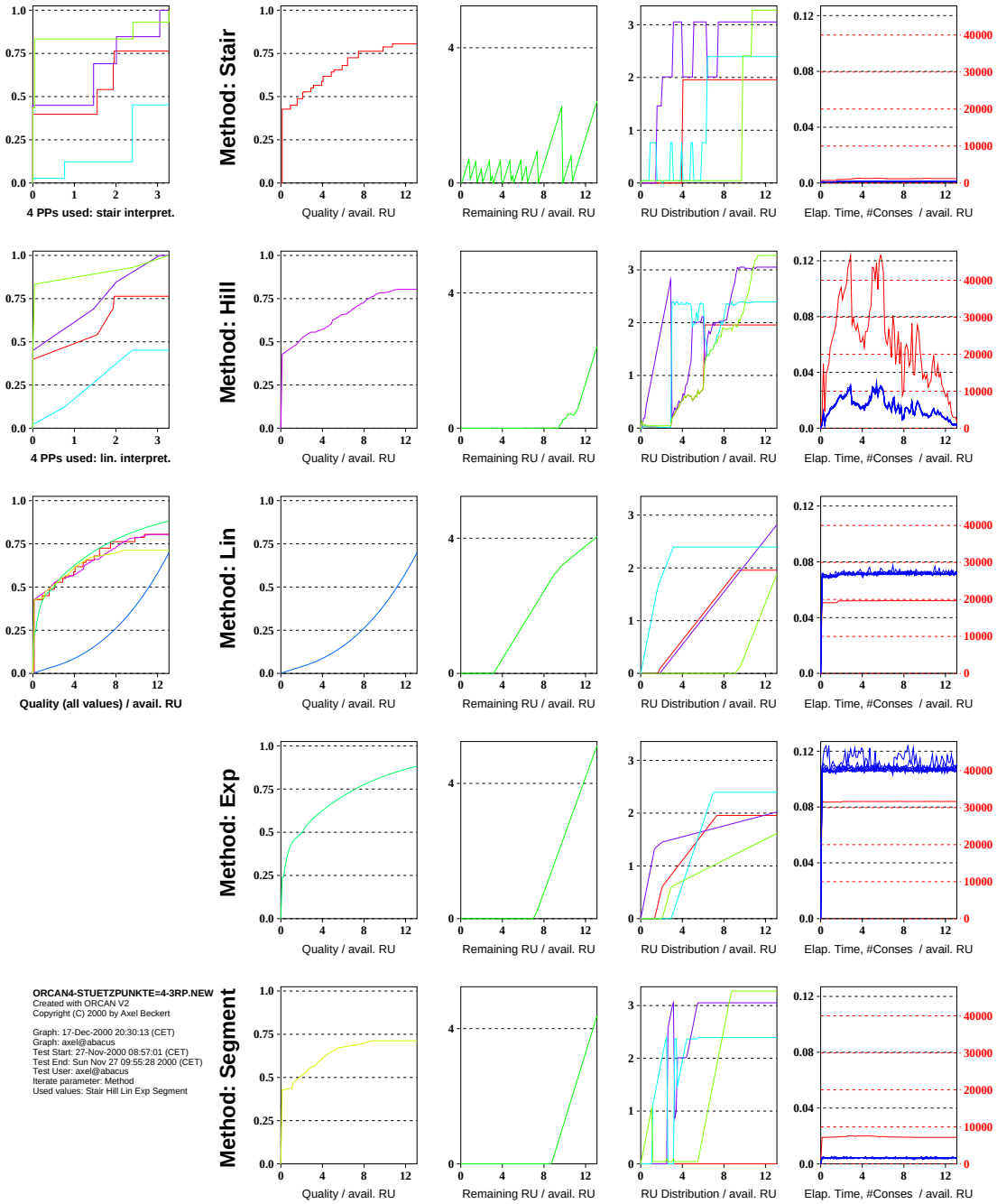

Abbildung 6.6.: Laufzeittest mit 4 Performanzprofilen mit je 4 Stützpunkten und Iteration über alle Kompilierungsmethoden

manzprofil-Format verwendet werden kann.

Die beiden regressiven Kompilierungsmethoden können jeweils mit zwei verschiedenen Arten von Performanzprofilen arbeiten: Sie akzeptieren Stützpunktlisten und die jeweiligen Parameter für ihre Funktionsklasse. Andere Eingabeformate ergeben bei ihnen — wie in Abschnitt 5.3 beschrieben — keinen großen Sinn. Die abschnittsweise lineare Kompilierungsmethode kann ebenfalls mit denselben Eingabeformaten wie die linear-regressive arbeiten, dazu müssen die Funktionsparameter allerdings noch in Stützpunktlisten umgewandelt werden.

Bezüglich des Eingabeformates weniger flexibel ist die Treppenstufen-Methode. Ihre starke Spezialisierung auf Stützpunktlisten läßt keine sinnvolle Möglichkeit zur Verwendung eines anderen Performanzprofil-Formates.

6.5. Analyse der Verwendbarkeit der Methoden in komplexen Anytime-Systemen

6.5.1. Eignung als Anytime-Algorithmus

Für die Implementation als Anytime-Algorithmen in ORCAN V2 eigneten sich besonders die Hillclimbing-Methode und das Treppenstufen-Verfahren: Der Hillclimbing-Algorithmus ist als Näherungsverfahren sehr leicht in einen Anytime-Algorithmus umwandelbar, da lediglich ein weiteres Abbruchkriterium neben der Genauigkeit des Ergebnisses eingebaut werden muß. Je später das Abbruchsignal kommt, desto besser ist die errechnete Verteilung, die beim Hillclimbing-Algorithmus immer die gesamte zu vergebende Ressourcenmenge umfaßt.

Bei der Treppenstufen-Methode sieht es ähnlich aus. Sie ist ebenfalls ein iteratives Verfahren, bei dem der Algorithmus nach jedem Iterationsschritt abgebrochen werden kann. Allerdings sind bei einem vorzeitigen Abbruch durch das Scheduling-System des übergeordneten Anytime-Moduls noch nicht alle verfügbaren Ressourcen verteilt. Dafür aber sind die bereits vergebenen Ressourcen auf diejenigen Anytime-Algorithmen verteilt, welche mit den zugeteilten Ressourcenmengen am effizientesten arbeiten.

Die drei übrigen Kompilierungsmethoden bauen Binärbäume für die Berechnung der Ressourcen auf, und erst nach Auf- und Abbau des Binärbaumes steht ein Ergebnis zur Verfügung. Bei einem Abbruch nach dem Aufbau, aber vor dem vollständigen Abbau, müssen bei einem Abbruchsignal die bisher aufgeteilten Ressourcenmengen

in einem extrem schnellen Verfahren (hier: Gleichverteilung) auf deren Subkomponenten aufgeteilt werden. Doch auch diese Berechnungen brauchen zusätzliche Rechenressourcen, die im schlimmsten Fall auch das auf diese Weise erzwungene Ergebnis zu spät liefern.

6.5.2. Analyse der Parameter der anytime-fähigen Methoden

In diesem Abschnitt werden verschiedene Werte für die Parameter der in Anytime-Systemen gut verwendbaren Kompilierungsmethoden miteinander verglichen, da diese das Verhalten der einzelnen Methoden und damit auch die Güte der Ergebnisse und den Ressourcenverbrauch zum Teil maßgeblich beeinflussen.

6.5.2.1. Analyse der Parameter der Hillclimbing-Methode

Parameter `:combine` Bezüglich des Parameters `:combine` der Hillclimbing-Methode zeigten die Laufzeittests sehr unterschiedliche Ergebnisse. Grund hierfür ist, daß bei der Multiplikation als Verknüpfungsfunktion Performanzprofile mit zu Beginn geringem oder keinem Qualitätszuwachs die Gesamtqualität stark senken bzw. sogar auf eine Qualität von 0 drücken.

Waren in den für den Laufzeittest verwendeten Performanzprofilen solche, deren Maximum weit unter einer Qualität von 1 lagen, so zeigte sich dies im Gesamtprofil durch einen sehr flachen Qualitätsverlauf. Dieser war teils so flach, daß die Toleranzgrenze des Hillclimbing-Algorithmus für die Qualitätssteigerung bereits bei den anfänglichen Ressourcenverteilungen unterschritten wurde und somit diese nach sehr kurzer Rechendauer und wenig Speicherverbrauch als Ergebnis zurückgegeben wurde. In den Diagrammen der Ressourcenverteilung zeigt sich dies durch den anfänglich gemeinsamen Verlauf aller Graphen. (Siehe Abbildungen A.9, A.10 und A.13.) In Abbildung A.9 ist die Qualitätssteigerung so niedrig, daß der Unterschied zur r-Achse des Diagramms erst durch eine Vergrößerung des entsprechenden Ausschnitts erkannt werden kann.

Anderseits gab es aber auch Fälle, bei denen die mit der Multiplikation als Verknüpfungsfunktion errechneten Ressourcenverteilungen stärker dem mit der Addition als Verknüpfungsfunktion errechneten Ergebnis als der Startverteilung ähnelten (siehe Abbildungen A.11 und A.12). Doch auch hier ist eindeutig erkennbar, daß bei wenig zu verteilenden Ressourcen die errechnete Verteilung der Startverteilung sehr nahe kommt.

Zwischen diesen beiden Extremen liegt der in Abbildung 6.7 gezeigte Laufzeittest,

da bei der Multiplikation als Verknüpfungsfunktion die Qualität nur mäßig steigt und die Ressourcenverteilung größtenteils der anfänglichen Verteilung entspricht. An den Stellen, an denen sie davon abweicht, zeigt sich deutlich der Zusammenhang zwischen dem Abweichen der Ressourcenverteilung von der Startverteilung und dem Speicherverbrauch bzw. der Laufzeit. Die entsprechenden Stellen sind in Abbildung 6.7 durch einen roten Kreis markiert.

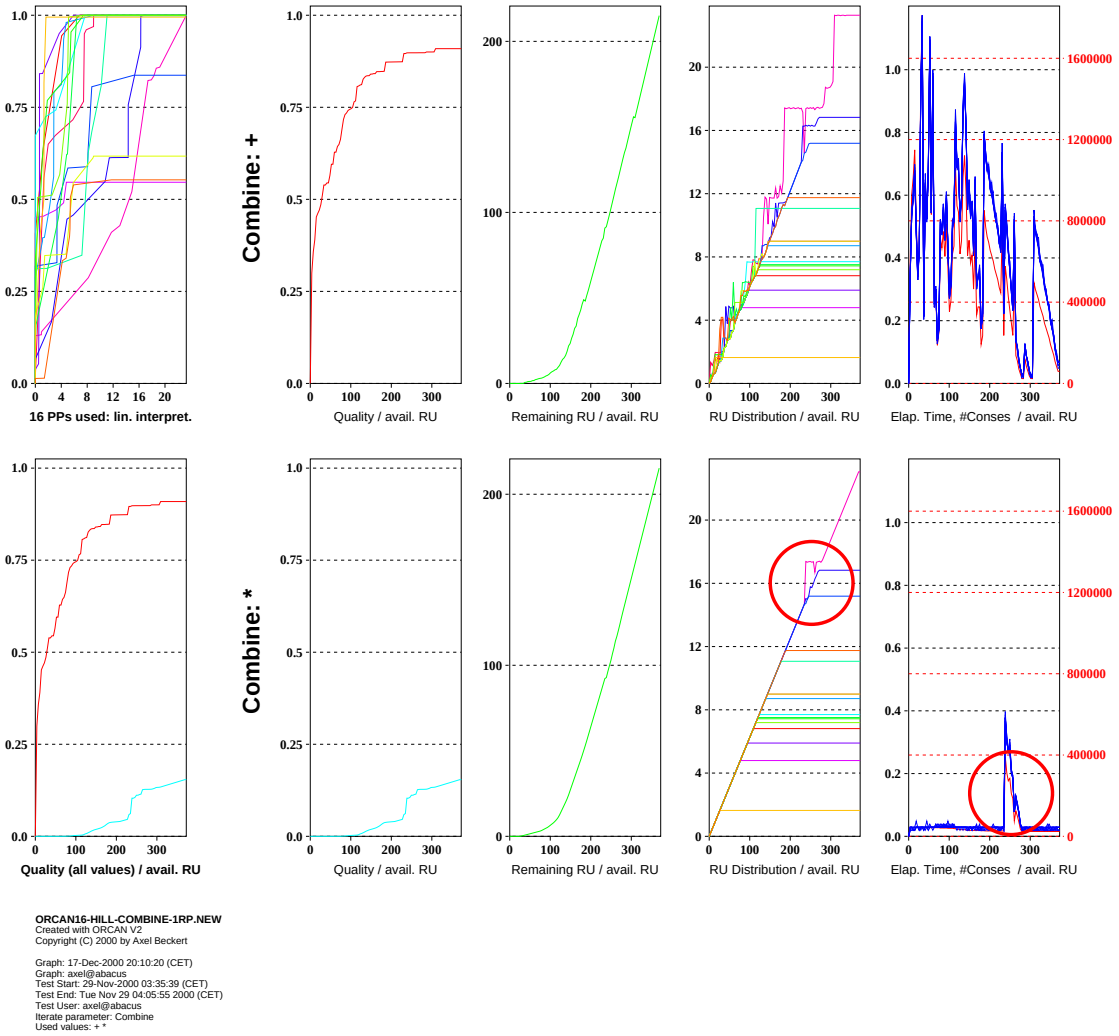

Abbildung 6.7.: Laufzeittest mit 16 Performanzprofilen und Iteration des Parameters `:combine` über die Werte `#'*` und `#'+`

Durchgängig zeigte sich bei allen Laufzeittests mit der Multiplikation als Verknüpfungsfunktion, daß die zu erwartende Qualität zu Beginn wesentlich flacher ist als die Varianten mit der Addition als Verknüpfungsfunktion, jedoch dafür gegen Ende stärker steigt. Das Diagramm eines Qualitätsverlaufes mit der Multiplikation

als Verknüpfungsfunktion tendiert also eher zu einer konvexen Form, während die Diagramme von Qualitätsverläufen mit der Addition als Verknüpfungsfunktion eher konkaven Charakter haben. Deutlich erkennbar ist dies in den Abbildungen 6.4 und A.12.

Parameter `:logical-operator` Die Verknüpfung der Abbruchbedingungen „Erreichen der unteren Grenze für die Schrittweite“ und „Erreichen der unteren Grenze für die Qualitätsverbesserung“ des Hillclimbing-Algorithmus durch *und* oder *oder* (Parameter `:logical-operator`) zeigte in den Laufzeittests nichts Unerwartetes: Die Verknüpfung mit *und* ergab häufig geringfügig bessere zu erwartende Qualitäten durch größere Abweichungen von der anfänglichen Verteilung gegenüber der Verknüpfung mit *oder*. Diese besseren Verteilungen mußten aber immer durch einen wesentlich höheren Speicherverbrauch und eine längere Laufzeit bezahlt werden.

In Abbildung 6.8 sind Speicherverbrauch und Laufzeit der Variante mit der Verknüpfung der Abbruchbedingungen mit *und* etwa zwei bis dreimal so hoch wie bei der Variante mit *oder* als Verknüpfung. Die Qualitätsverbesserung gegenüber dieser Variante beträgt aber nur wenige Prozent. Dies spiegelt sich auch in den Abbildungen A.14, A.15 und A.16 im Anhang wieder.

Parameter `:tolerance` Es zeigte sich durchgehend, daß die Wahl eines zu großen Wertes des Hillclimbing-Methoden-Parameters `:tolerance` bereits bei der initialen Verteilung zu einer Erfüllung der Abbruchbedingung führen und eine zu kleine Wahl sowohl Laufzeit als auch Speicherverbrauch sehr stark in die Höhe treiben, aber keine allzu großen Verbesserungen in der Verteilung bzw. Qualität mit sich bringen. Siehe hierzu die letzten drei Qualitätsverläufe in Abbildung 6.9. Sie unterscheiden sich kaum noch, obwohl Laufzeit und Speicherverbrauch weiter steigen. In diesem Beispiel wäre wohl je nach Einsatzzweck und Ressourcenbeschränkungen (falls ORCAN selbst als Anytime-Algorithmus läuft) ein Wert irgendwo zwischen 0,01 und 0,0001 sinnvoll. In Vergleich zur Ressourcenmenge, die notwendig ist, damit alle verwendeten Performanzprofile ihren Maximalwert erreichen[11], entspricht dies etwa dem $5 \cdot 10^{-5}$- bis $5 \cdot 10^{-7}$-fachen dieser Menge.

Parameter `:always-test-dir` Die Ergebnisse bezüglich des Hillclimbing-Methoden-Parameters `:always-test-dir` fallen wie erwartet sehr eindeutig aus: Wie in den Abbildungen 6.10 und A.21 (Seite 128) deutlich zu erkennen ist, bringt das Testen der „Richtung“ in jedem Iterationsschritt nur wenig Qualitätsverbesserung,

[11] In Abbildung 6.9 liegt dieser Wert zum Beispiel um die 200 Ressourceneinheiten.

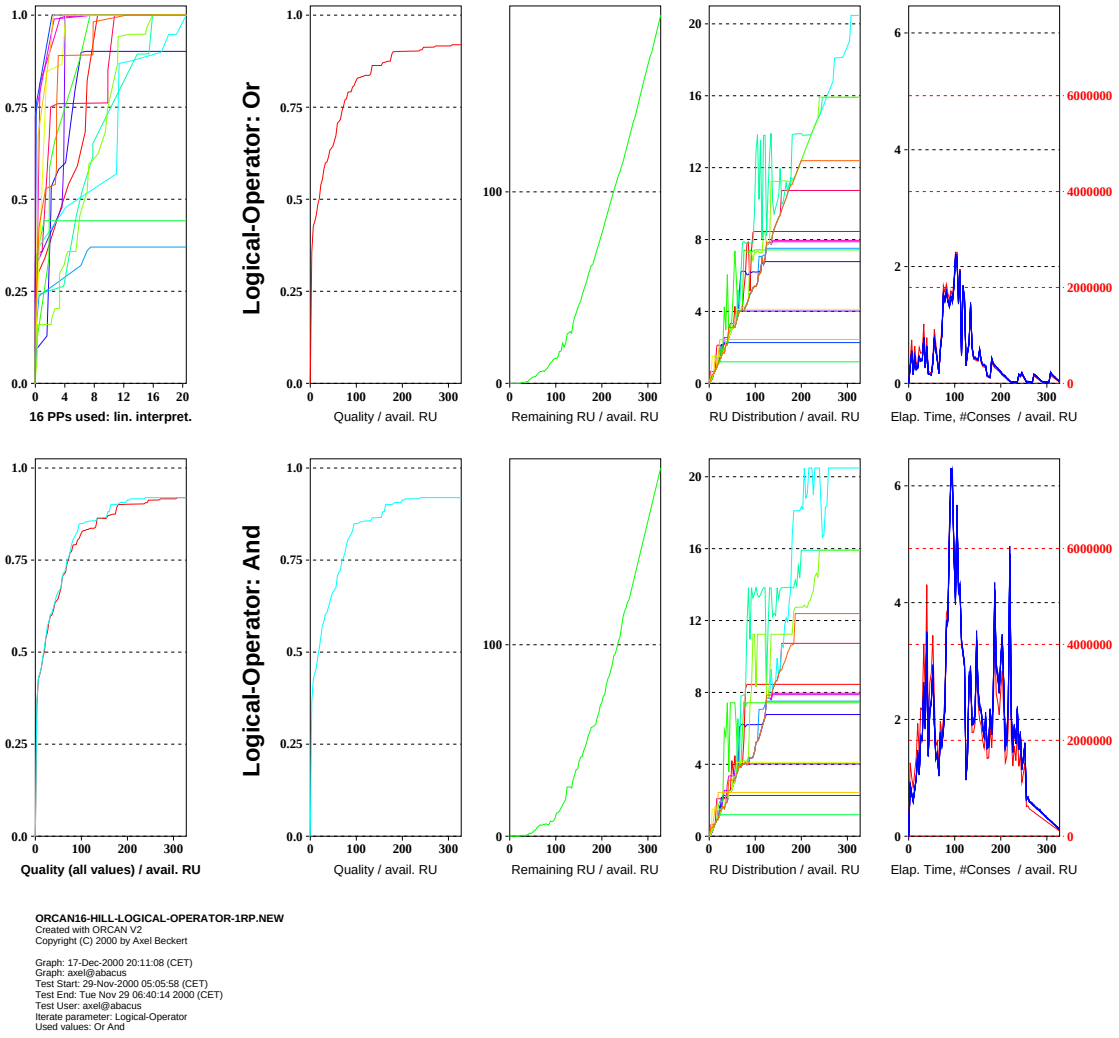

Abbildung 6.8.: Laufzeittest mit 16 Performanzprofilen und Iteration des Parameters `:logical-operator` über die Werte `#'and` und `#'or`

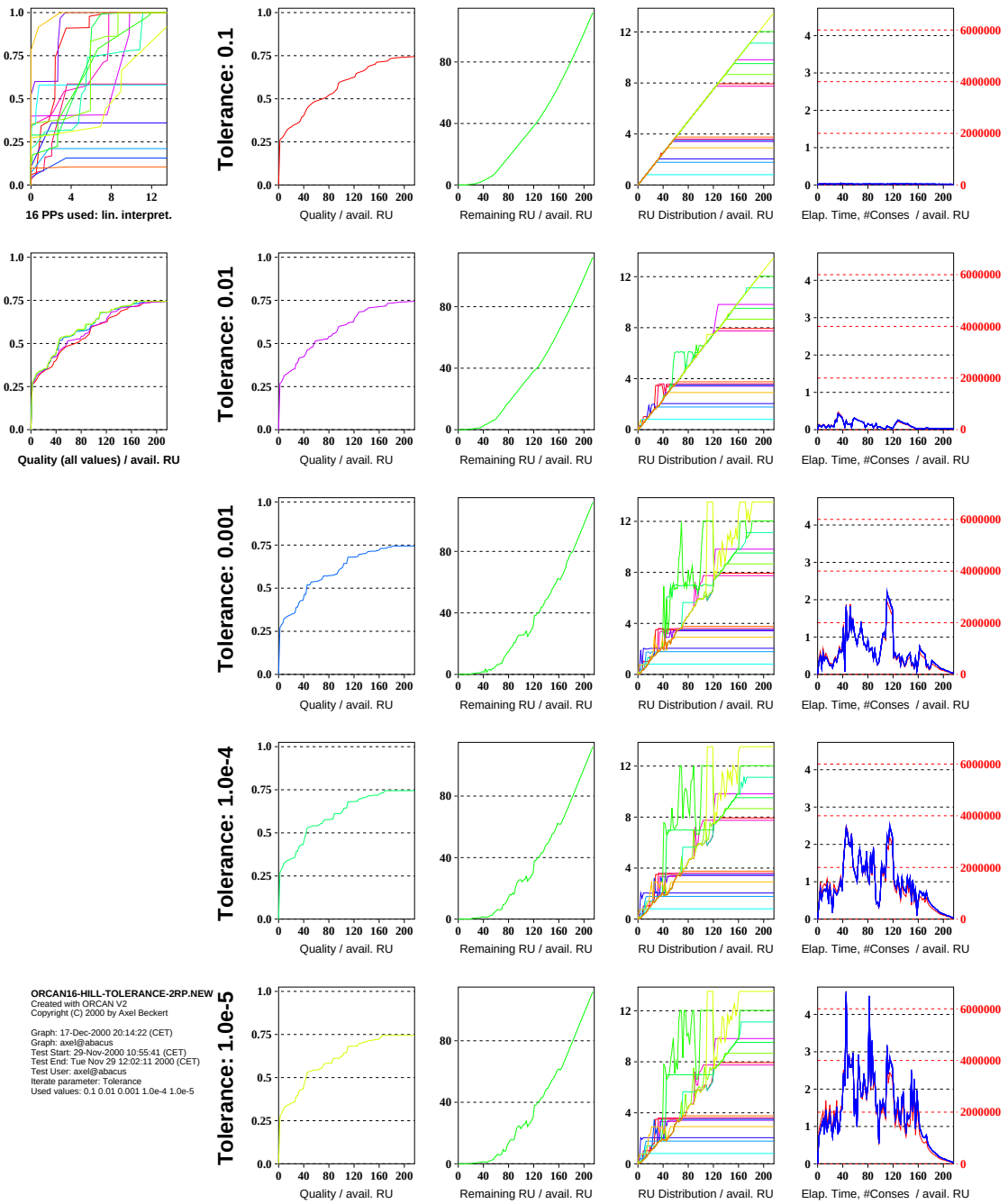

Abbildung 6.9.: Laufzeittest mit 16 Performanzprofilen und Iteration des Hill-climbing-Algorithmus-Parameters `:tolerance` über die Werte 10^{-n} mit $n \in \{1, \ldots, 5\}$

braucht dafür aber wesentlich mehr Rechenressourcen: Sowohl Laufzeit als auch Speicherverbrauch stiegen auf die drei- bis achtfachen Werte.

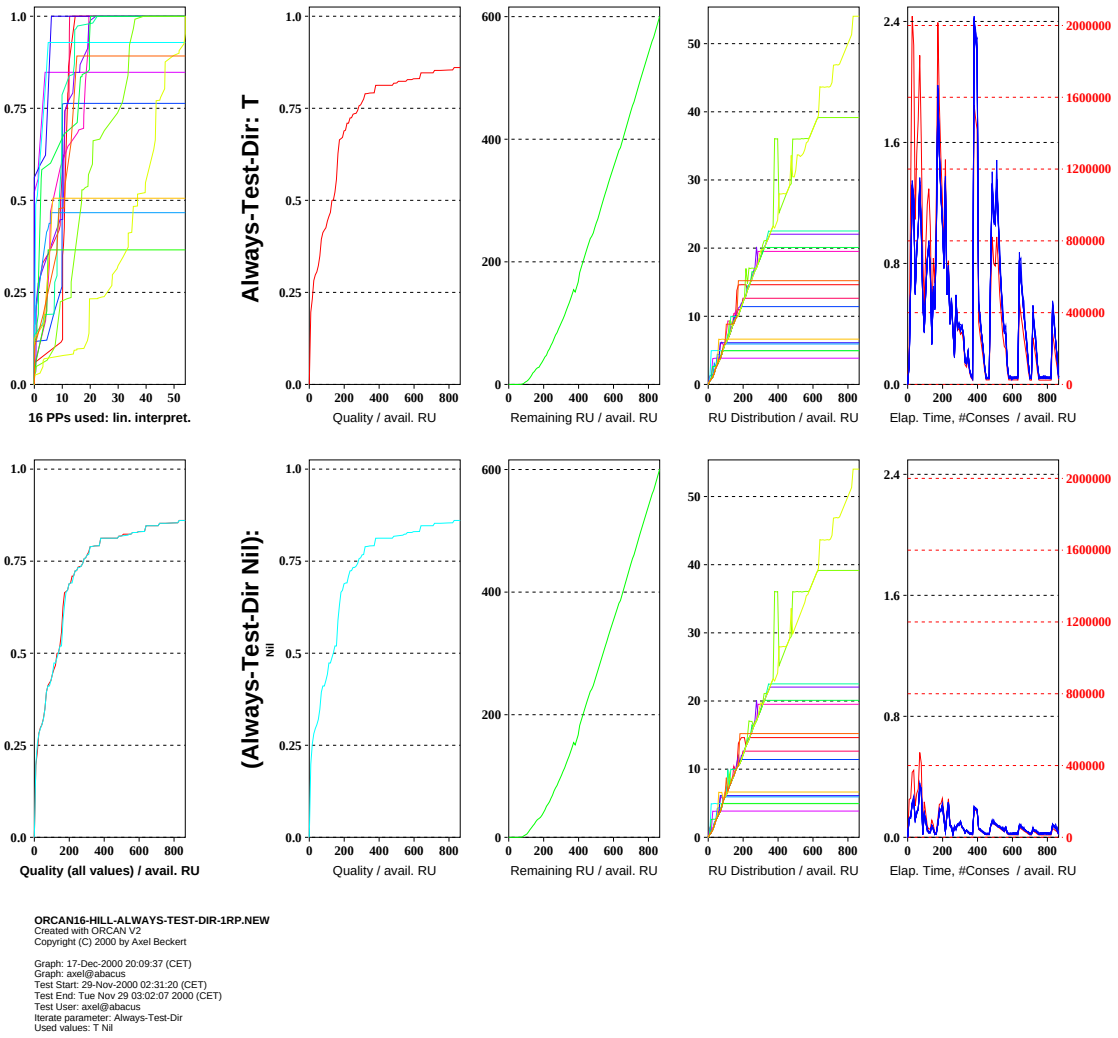

Abbildung 6.10.: Laufzeittest mit 16 Performanzprofilen und Iteration des Parameters `:always-test-dir` über die Werte `t` und `nil`

6.5.2.2. Analyse der Parameter der Treppenstufen-Methode

Im folgenden Abschnitt werden die möglichen Varianten zur Mehrverteilung von Ressourcen, wie sie in den Abschnitten 4.3.2.4 und 5.5 beschrieben wurden, miteinander verglichen. Dabei gelten die gemachten Beobachtungen sowohl für Varianten mit einer Beschränkung der Ressourcenvergabe auf die doppelte Menge der

ursprünglich zu vergebende Ressourcenmenge (Parameter $\alpha = 1$ und $\beta = 0$) als auch für Varianten mit einer Beschränkung auf die ursprünglich zu vergebenden Ressourcenmenge plus der bisher noch nicht verteilten Ressourcenmenge (Parameter $\alpha = 0$ und $\beta = 1$).

Der darauffolgende Abschnitt vergleicht dagegen, wie sich die unterschiedlichen Beschränkungen auf den Verbrauch von Rechenressourcen und das Ergebnis auswirken.

Parameter `:type` In Abbildung 6.11 ist anhand der negativen Werte in den Restressourcendiagrammen deutlich erkennbar, wie mehr Ressourcen verteilt wurden, als anfänglich zu vergeben waren.

Bei `:type :none` (keine Mehrvergabe) gibt es keine negativen Restressourcen.

Die nächsten beiden Varianten sind sehr ähnlich, der Unterschied liegt darin, daß die Beschränkung der Mehrvergabe auf die doppelte Menge der ursprünglich zu vergebenden Ressourcenmenge nur bei `:type :abmt` wirksam ist. Einen Unterschied gibt es nur zu Beginn (siehe die roten Kreise).

Die letzten beiden Varianten sind sich ebenfalls ähnlich: Bei `:type :abqu` wird innerhalb der gegebenen Beschränkung die größte Qualitätssteigerung gewählt, bei `:type :abgr` die effizienteste. Entsprechend wird bei `:type :abgr` an vielen Stellen eine geringere Mehrvergabe gemacht, als bei `:type :abqu` (siehe die blauen Kreise). Dafür gibt es bei `:type :abqu` eine höhere Qualität an diesen Stellen (siehe Unterschied zwischen gelbem und grünen Graph im violetten Kreis).

Parameter `:alpha` **und** `:beta` Bezüglich der Parameter `:alpha` und `:beta` der Treppenstufen-Methode kann nur schwer eine generelle Empfehlung gegeben werden, da sinnvolle Werte für diese beiden Parameter stark von der Art des geplanten Einsatzes abhängen:

Ein Wert von $\beta > 0$ (in Abbildung 6.12 wurden $\alpha = 0$ und $\beta = 1$ verwendet) eignet sich gut, wenn eine Mehrvergabe in geringen Mengen möglich, aber nicht unbedingt notwendig ist; die Restressourcen schwanken um den Wert 0 und die Beträge von positiven und negativen Restressourcen bewegen sich im selben Bereich.

Ein Wert von $\alpha > 0$ (in Abbildung 6.12 wurden $\alpha = 1$ und $\beta = 0$ verwendet) führt dagegen — sofern noch nicht alle Anytime-Algorithmen ihre maximal mögliche Qualität erreicht haben — fast immer zu einer Mehrvergabe, die Restressourcen sinkt mit der Erhöhung der zur Verfügung stehenden Ressourcen weit in den negativen Bereich. Erst, wenn durch die Erhöhung der zu Verfügung stehenden Ressourcen kei-

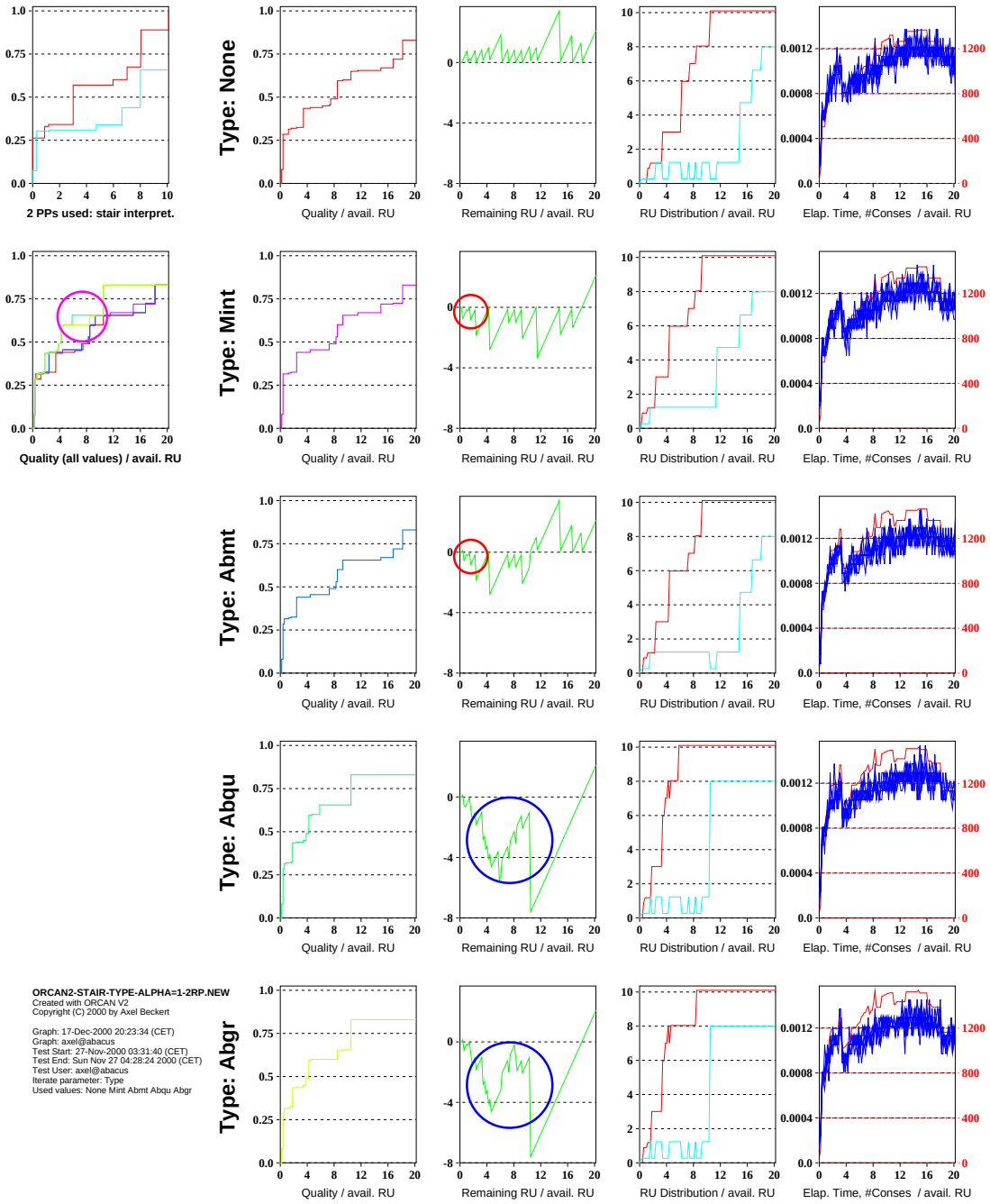

Abbildung 6.11.: Laufzeittest mit 2 Performanzprofilen und Iteration des Treppenstufen-Methode-Parameters `:type` (bei $\alpha = 1$ und $\beta = 0$) über sämtliche möglichen Werte

ne Verbesserung der Qualität mehr erreicht wird, steigen die Restressourcen wieder in den positiven Bereich.

Abbildung 6.12 zeigt in der Restressourcen-Spalte deutlich die Auswirkungen der beiden betrachteten Parameter α und β.

6.6. Bewertung und Ergebnisse der Analyse

Es gibt verschiedene Fälle, in denen nur bestimmte Kompilierungsmethoden verwendet werden können. In mehreren dieser Fälle hängt dies davon ab, wie kompliziert die Verknüpfungen zwischen einzelnen Anytime-Algorithmen sind und ob sie bei der Ressourcenverteilung beachtet werden sollten.

Nicht-triviale Verknüpfungen der Anytime-Algorithmen untereinander lassen sich im Normalfall nur durch Lambda-Ausdrücke darstellen und sind damit nur mit der Hillclimbing-Methode handhabbar. Doch selbst wenn die Verknüpfungen trivial sind[12], kann sich die genaue Anzahl der Anytime-Algorithmen negativ auf ihre Gleichberechtigung auswirken: Die beiden regressiven sowie die abschnittsweise lineare Kompilierungsmethoden bauen zur Ressourcenverteilung einen Binärbaum auf und können die Gleichberechtigung aller verwendeten Anytime-Algorithmen nur dann gewähren, falls die Anzahl der verwendeten Anytime-Algorithmen eine Zweierpotenz ist.

Bei vielen Performanzprofilen empfiehlt sich die Treppenstufen-Methode, da ihr Rechenaufwand nur linear mit der Anzahl der Performanzprofile steigt. Selbst die niedrigsten Laufzeitwerte der Hillclimbing-Methode liegen meist über denen der Treppenstufen-Verfahren. Einziger Kritikpunkt an der Treppenstufen-Methode hier: In bestimmten, aber seltenen Fällen, kann sich ihr Ressourcenverbrauch vervielfachen (siehe Seite 43).

Da der Ressourcenverbrauch der restlichen drei Kompilierungsmethoden mit der Anzahl der Performanzprofile exponentiell steigt, und weil sich durch die Binärbaum-Struktur auch die Fehler der Approximationen aufschaukeln können, sind diese Methoden im allgemeinen nicht zu empfehlen.

Auch bei wenigen Performanzprofilen ist die Treppenstufen-Methode aufgrund ihres niedrigen Ressourcenverbrauchs und ihrer nicht-approximierten Rechenweise die zu empfehlende Methode. Die Hillclimbing-Methode zeigt — wie in fast allen Fällen — starke Schwankungen im Ressourcenverbrauch, ihre Ergebnisse entsprechen aber bis

[12] Hierzu zählen u. a. die Addition, die Multiplikation und das arithmetische Mittel.

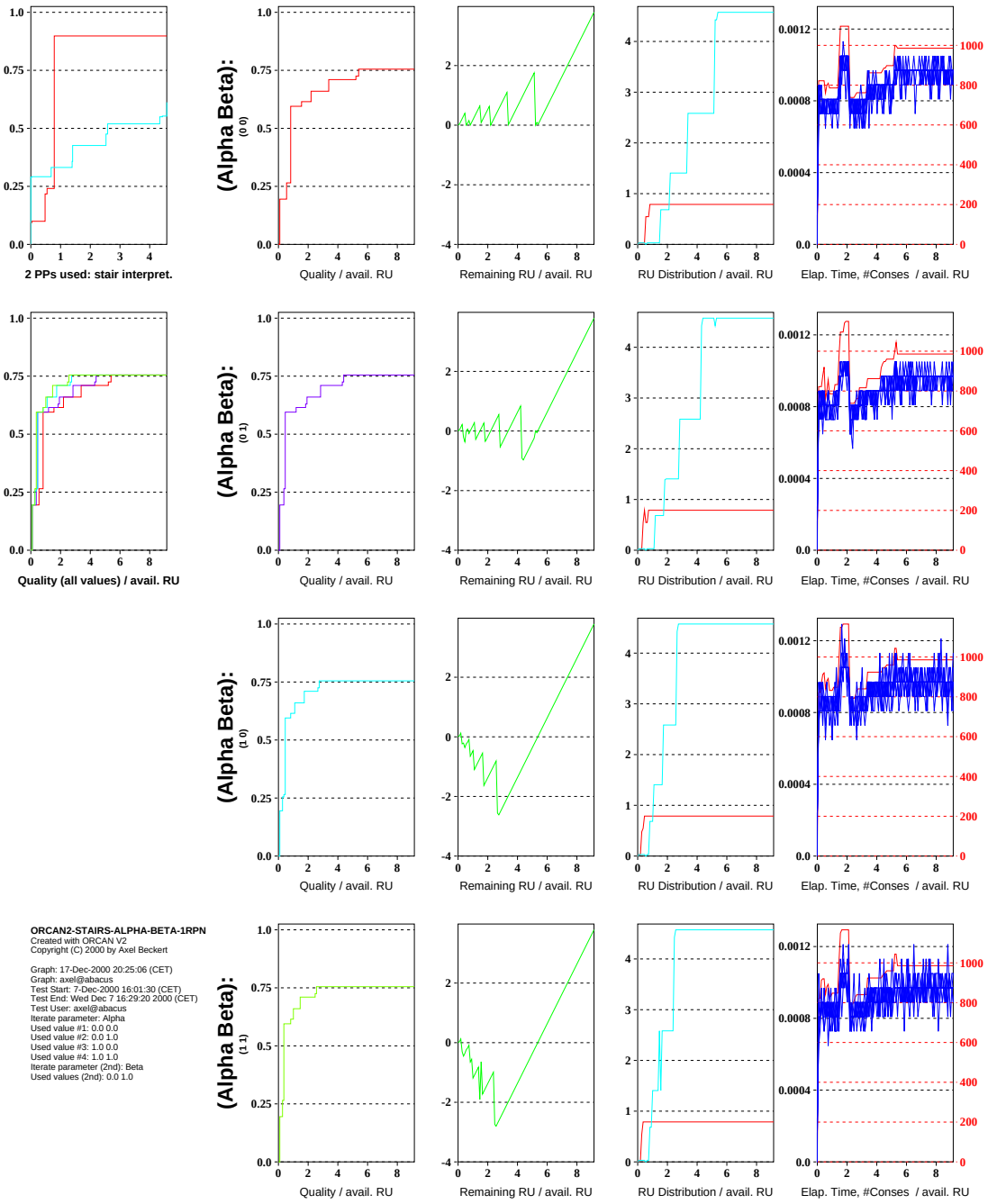

Abbildung 6.12.: Laufzeittest mit 2 Performanzprofilen und Iteration der Parameter `:alpha` und `:beta` der Treppenstufen-Methode jeweils über die Werte 0 und 1

auf die unterschiedlichen Interpretationsmethoden ungefähr denen der Treppenstufen-Methode. Negativ fällt auch die exponentiell-regressive Kompilierungsmethode auf, da sie im Vergleich zur linear-regressiveen und zur abschnittsweise linearen Kompilierungsmethode wesentlich mehr Rechenressourcen braucht.

Sind die Performanzprofile in unterschiedlichen Eingabeformaten gegeben, so bietet sich insbesondere die Hillclimbing-Methode an, da sie Performanzprofile aller Art verarbeiten kann.

Bei im Vergleich zu den r-Werten der Performanzprofile kleinen zu vergebenden Ressourcenmengen sind nur Kompilierungsmethoden, die nicht die Multiplikation als Verknüpfungsfunktion benötigen, zu verwenden, da diese im gegebenen Fall oft nur eine zu erwartende Qualität von Null berechnen. Bei großen Ressourcenmengen dagegen fällt nur die Hillclimbing-Methode positiv auf, da sie weniger Ausreißer im Ressourcenverbrauch als sonst hat.

Bei vielen Stützpunkten ist von der abschnittsweisen Kompilierungsmethode abzuraten, während sie bei wenigen Stützpunkten eine Alternative zur Treppenstufen-Methode darstellt. Ist die Anzahl der Stützpunkte hoch, ist die Treppenstufen-Methode trotz der maximal quadratischen Abhängigkeit ihres Ressourcenverbrauchs von der Anzahl der Stützstellen die beste Wahl, da sie bei nach wie vor niedrigem Ressourcenverbrauch sehr differenzierte Ressourcenverteilungen liefert.

Geht es um kurze Laufzeiten — beispielsweise bei Kompilierungen in Echtzeit — oder geringen Speicherverbrauch, so ist eindeutig das Treppenstufen-Verfahren vorzuziehen, da es in fast allen Fällen die schnellste und sparsamste Kompilierungsmethode ist. Die häufig zweitschnellste Kompilierungsmethode, die abschnittsweise lineare, hat wesentlich mehr Nachteile als die Treppenstufen-Methode und stellt daher keine besonders gute Alternative dar.

Die Hillclimbing-Methode glänzt zwar teilweise ebenfalls mit sehr kurzen Laufzeiten und geringem Speicherverbrauch, hat aber auch Spitzenwerte im entgegengesetzten Extrem. Insgesamt sind sowohl Laufzeit als auch Speicherverbrauch der Hillclimbing-Methode sehr schlecht im voraus abschätzbar, weswegen sie für ressourcensensible Anwendungen nicht in Frage kommt. Da bei der Hillclimbing-Methode außerdem die Gefahr besteht, daß sie sich in einem lokalen Extremum verfängt, ist sie ausschließlich für Anwendungen zu empfehlen, bei denen Lambda-Ausdrücke als Eingabeformat notwendig sind (siehe oben).

Kommt es darauf an, daß die zu erwartende Qualität relativ genau vorhergesagt werden muß, so fallen die regressiven Kompilierungsmethoden aufgrund der Approximation der Performanzprofile in die entsprechenden Funktionsklassen weg, da die von ihnen errechneten Qualitätswerte vor allem bei sehr kleinen oder großen

Ressourcenwerten häufig sehr stark von den zu erwartenden Qualitätswerten abweichen, insbesondere auch nach oben. Die abschnittsweise lineare Methode berechnet zwar — wie die Hillclimbing-Methode und das Treppenstufen-Verfahren — die zu erwartende Qualität anhand der Stützpunktlisten korrekt, jedoch liegt die durch berechnete Ressourcenverteilung erreichbare Qualität besonders bei großen zu verteilenden Ressourcenmengen häufig wesentlich unter den erreichbaren Werten. Insofern ist in diesem Fall nur das Treppenstufen-Verfahren, bedingt (siehe oben) auch die Hillclimbing-Methode zu empfehlen.

In Tabelle 6.1 sind die Eignungen, Vor- und Nachteile der verschiedenen Kompilierungsmethoden noch einmal zusammengefaßt. Die Bewertung erfolgt mit Schulnoten (siehe Tabelle 6.2). Die angegebenen Gesamtnoten errechnen sich aus dem arithmetischen Mittel der Einzelnoten, wobei jede Note so häufig, wie in der Spalte „Faktor“ angegeben, in die Wertung eingeht. Die dabei verwendete Bewertung muß nicht für sämtliche möglichen Einsatzzwecke passend sein.

6.7. Verwendung der Ergebnisse in ressourcenadaptierenden Systemen

Sollen die im vorigen Abschnitt zusammengefaßten Ergebnisse in die Erzeugung von System-Performanzprofilen in Form einer automatischen Auswahl der passenden Kompilierungsmethode und Parameter einbezogen, also ein ressourcenadaptierendes System konstruiert werden, so bietet sich die folgende Vorgehensweise an:

Zuerst wird anhand der Eingabeformate der gegebenen Performanzprofile und der in den Abschnitten 5.3 und 6.4.3 erwähnten Restriktionen entschieden, welche Kompilierungsmethoden überhaupt möglich sind. Dies kann beispielsweise in einer Positivliste gespeichert werden. Danach sollte — falls analysierbar — die geplante Verknüpfungsfunktion auf nicht-triviale Verknüpfung der Parameter untersucht und die Positivliste gegebenenfalls weiter eingeschränkt werden.

Sind danach noch mehr als eine Kompilierungsmethode in der Positivliste, so sollte anhand von Eigenschaften wie der Anzahl der Performanzprofile und durchschnittlichen Anzahl von Stützpunkten entschieden werden, welche der Methoden in der Positivliste schließlich verwendet wird.

Handelt es sich bei dem umgebenden System um ein Anytime-System, so sollte an dieser Stelle auch darauf geachtet werden, wie hoch der Ressourcenverbrauch der in der Positivliste vorhandenen Kompilierungsmethoden bei den entsprechenden Anzahlen von Stützpunkten und Performanzprofilen ist.

Fall oder Eigenschaft / Faktor		LinReg	ExpReg	AbsLin	Hill	Treppen
Nicht-triviale Verknüpfungen der Anytime-Algorithmen untereinander	4	5	5	5	1	4
Gleichberechtigung der Performanzprofile abhängig von $\lvert\mathbb{P}\rvert \overset{?}{\in} \{2^n \lvert n \in \mathbb{N}\}$ (Ja=5, Nein=1)	3	5	5	5	1	1
Qualität bei genügend Ressourcen entspricht der maximal möglichen Qualität	2	3	4	5	1	1
Laufzeit	8	4	4	2	3	1
Speicherverbrauch	6	4	4	2	3	1
Vorhersagbarkeit von Laufzeit und Speicherverbrauch	5	2	2	2	6	1
Hohe Menge an zu verteilenden Ressourcen	1	3	3	3	2	3
Niedrige Menge an zu verteilenden Ressourcen	1	4	4	3	1	2
Viele Performanzprofile ($\lvert\mathbb{P}\rvert > 30$)	2	4	4	5	2	1
Wenige Performanzprofile ($\lvert\mathbb{P}\rvert < 5$)	1	3	5	3	3	2
Performanzprofile mit vielen Stützstellen ($\lvert\mathbb{S}_i\rvert > 100$)	2	4	4	6	3	1
Performanzprofile mit wenigen Stützstellen ($\lvert\mathbb{S}_i\rvert < 6$)	1	3	4	3	2	1
Heterogene Performanzprofile	1	3	3	3	1	6
Einfache Implementation als Anytime-Algorithmus	2	5	5	5	1	2
Gesamtnote:		**3,82**	**3,95**	**3,33**	**2,62**	**1,59**

Tabelle 6.1.: Tabellarischer Überblick

Note	Eignung	Eigenschaft	Verbrauch
1	sehr geeignet	sehr gut	sehr gering
2	geeignet	gut	gering
3	bedingt geeignet	befriedigend	niedrig
4	weniger geeignet	ausreichend	akzeptabel
5	schlecht geeignet	mangelhaft	hoch
6	ungeeignet	ungenügend	sehr hoch

Tabelle 6.2.: Beschreibung der „Schulnoten“ aus Tabelle 6.1

Abschließend sollten die methoden-spezifischen Parameter anhand der Größenordnung der r-Werte in den Stützpunktlisten (falls vorhanden) und der zu erwartenden Ressourcenbeschränkungen (aus von ORCAN unabhängigen Wissensquellen; falls vorhanden) festgelegt werden.

In ORCAN wäre die Funktion `#'spp` die geeignete Stelle, um solch eine Funktionalität zu implementieren.

7. Zusammenfassung und Aussichten

Ziele dieser Arbeit waren die Erstellung eines Systems zur Berechnung von Ressourcenverteilungen und System-Performanzprofilen, welches unterschiedliche Kompilierungsmethoden verwenden kann und flexibel einsetzbar ist, ferner die Verfeinerung bekannter sowie die Erarbeitung neuer heuristischer Kompilierungsmethoden inklusive deren Analyse. Als Grundlage wurden dabei der Anytime-Kompiler nach Zilberstein (1993) sowie die Implementation von ORCAN V1 [Baus & Beckert, 1998] verwendet, welche an vielen Stellen verbessert und verallgemeinert sowie umfangreich erweitert wurde.

Hervorzuheben ist hierbei insbesondere das Treppenstufen-Verfahren zur Berechnung von Ressourcenverteilungen. Es führt im Vergleich zu den bisher verwendeten Kompilierungsmethoden mehrere Neuerungen ein: Bisherige Kompilierungsmethoden (insbesondere der Hillclimbing-Algorithmus und die beiden regressiven Kompilierungsmethoden) würden auch auf nicht monotonen Stützpunktlisten arbeiten; die Treppenstufen-Methode dagegen ist auf monoton steigende Stützpunktlisten spezialisiert und ermöglicht so einen wesentlich angepaßteren Algorithmus zum Berechnen von Ressourcenverteilungen. Auch stellt es eine Verbesserung dar, daß es sich um einen bislang für Anytime-Kompiler nicht verwendeten Greedy-Algorithmus handelt, welcher garantieren kann, daß die bereits verteilten Ressourcen den entsprechenden Subkomponenten auf jeden Fall zur Verfügung stehen. Bei diesem Verfahren sind ebenfalls neu die Möglichkeit, mehr Ressourcen zu vergeben, als ursprünglich vorgesehen waren, sowie Aussagen über die beste Reihenfolge bei der Verwendung der Ressourcen machen zu können.

Die Option, die Ergebnisse von Laufzeitmessungen mitsamt der Ressourcenverteilungen grafischen darzustellen, erlaubt zudem eine unkomplizierte Analyse verschiedener Kompilierungsmethoden und Performanzprofil-Arten.

Endergebnis ist ein Verfahren, das eine wesentlich effizientere Kompilierung von Anytime-Algorithmen als bisher ermöglicht — sowohl durch eine neue, spezialisierte Kompilierungsmethode als auch durch Aussagen darüber, für welche Art von Einsatz sich welche Kompilierungsmethode mit welcher Parametrisierung am besten eignen.

Problematisch sind zur Zeit noch Anytime-Module, deren Subkomponenten nicht-trivial miteinander verknüpft sind. Ihre System-Performanzprofile können bisher nur mit einer Kompilierungsmethode berechnet werden, die beliebige Funktionen als Verknüpfungsfunktionen verwenden kann. Dies ist aktuell nur beim Hillclimbing-Algorithmus der Fall. Hier wäre eine Methode wünschenswert, welche wie die Treppenstufen-Methode durch eine Spezialisierung auf monoton steigende Stützpunktlisten wesentlich effizienter und bezüglich des Ressourcenverbrauchs besser voraussagbar als die Hillclimbing-Methode ist.

Ebenso ist noch unbekannt, inwiefern sich die Qualität der Ergebnisse durch Verkettung verschiedener Kompilierungsmethoden effizient verbessern läßt. Dabei erscheinen insbesondere zwei Varianten interessant:

1. Es ist möglich, den Startpunkt für eine Austauschheuristik wie dem Hillclimbing-Algorithmus durch eine der anderen Methoden berechnen zu lassen, um bereits zu Beginn der lokalen Suche näher am globalen Extremum zu sein und damit die Wahrscheinlichkeit zu senken, sich in einem lokalen Extremum zu verfangen.

2. Eine Austauschheuristik kann genauso dazu verwendet werden, das bereits relativ genaue Ergebnis einer anderen Kompilierungsmethode lokal zu optimieren, falls noch Ressourcen übrig sind. Dies scheint vor allem dann vorteilhaft, wenn das Wissen über die zur Verfügung stehenden Ressourcen nur sehr ungenau ist, da dann eine zu Beginn schnelle, gegebenenfalls etwas gröbere Kompilierungsmethode gewählt werden kann und deren Ergebnis durch eine Austauschheuristik in Form eines Anytime-Algorithmus solange verbessert wird, bis keine Ressourcen mehr übrig sind, d. h. der Anytime-Algorithmus das Signal zum Abbrechen bekommt.

Weitere interessante Fragen sind, inwiefern sich die Algorithmen der Kompilierungsmethoden parallelisieren lassen, und ob die informationsreicheren Ergebnisse der Treppenstufen-Methode auf parallelen Systemen zu ähnlichen Vorteilen (siehe Abschnitt 4.3.2.5) wie auf nicht-parallelen Systemen führen.

Ebenso interessant wäre, zu wissen, welche Kompilierungsalgorithmen auch auf die in Abschnitt 2.5.3 näher beschriebenen Performanzverteilungsprofile anwendbar sind bzw. wie stark die Algorithmen dazu abgeändert werden müssen.

Weiterführende Arbeiten könnten außerdem untersuchen, inwiefern sich andere heuristische Methoden zur Lösung kombinatorischer Optimierungsprobleme für die Kompilierung von Anytime-Algorithmen eignen. Insbesondere die der Natur abgeschauten Ansätze, wie z. B. die Evolutionsstrategie [Rechenberg, 1972; Schwe-

fel, 1977], Genetische Algorithmen [Goldberg, 1989] und Ant Colony Optimization (ACO) [Dorigo & Di Caro, 1996; Dorigo & Gambardella, 1998] sind vielversprechend.

A. Weitere Beispiele

Neben den im folgenden sowie in Kapitel 6 aufgeführten ORCAN Laufzeittests sind im Rahmen dieser Arbeit viele weitere Laufzeitmessungen zur statistischen Untermauerung der Ergebnisse durchgeführt worden. Da aber die Aufnahme der Diagramme aller durchgeführten Laufzeittests den Umfang dieser Arbeit bei weitem übersteigen würde, werden sie getrennt in [Beckert, 2001] veröffentlicht.

A.1. Vergleich der verschiedenen Kompilierungsmethoden bei unterschiedlicher Anzahl oder Art von Performanzprofilen

A.1.1. Verschiedene Anzahlen von Performanzprofilen

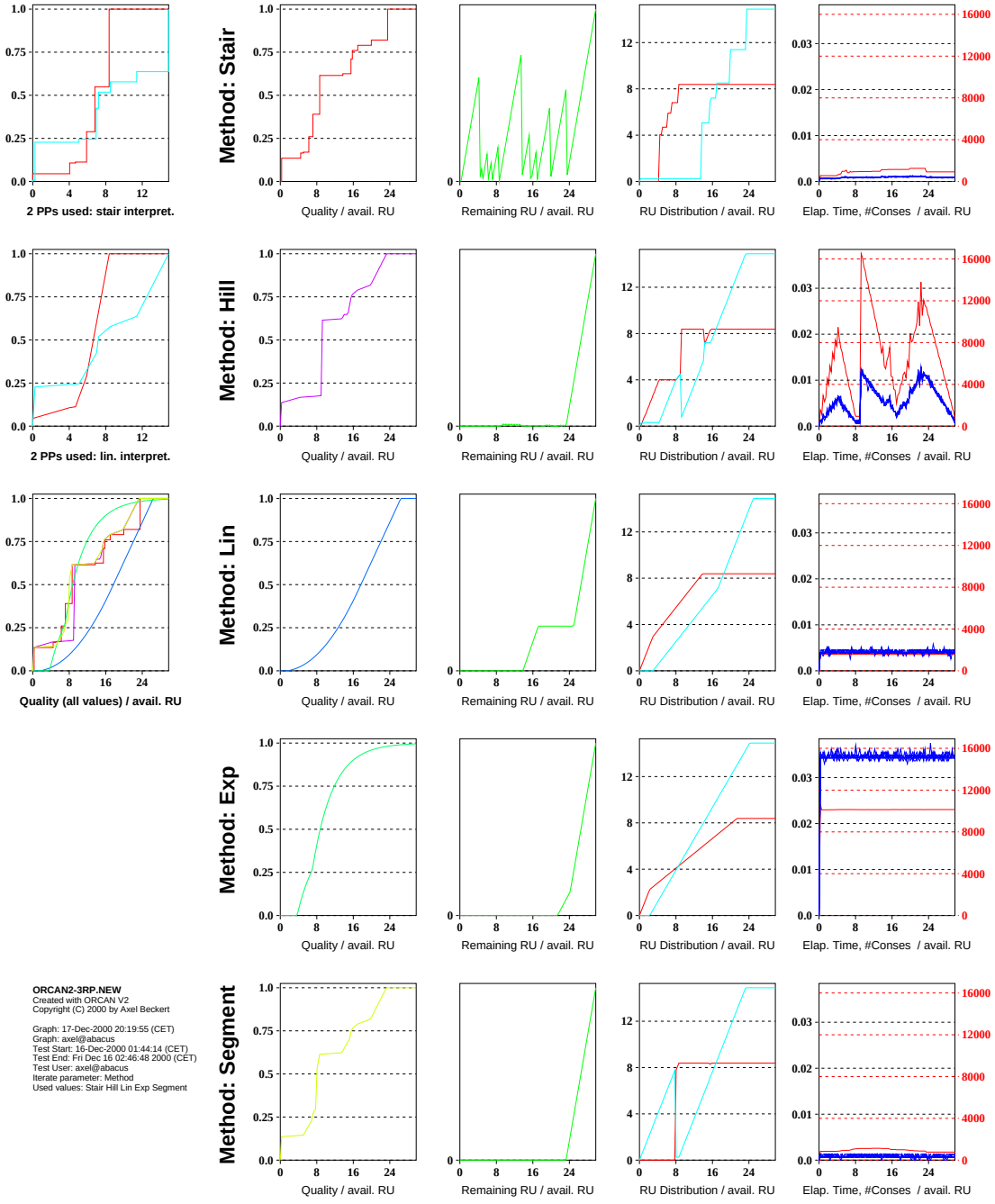

Abbildung A.1.: Laufzeittest mit 2 Performanzprofilen und Iteration über alle Kompilierungsmethoden

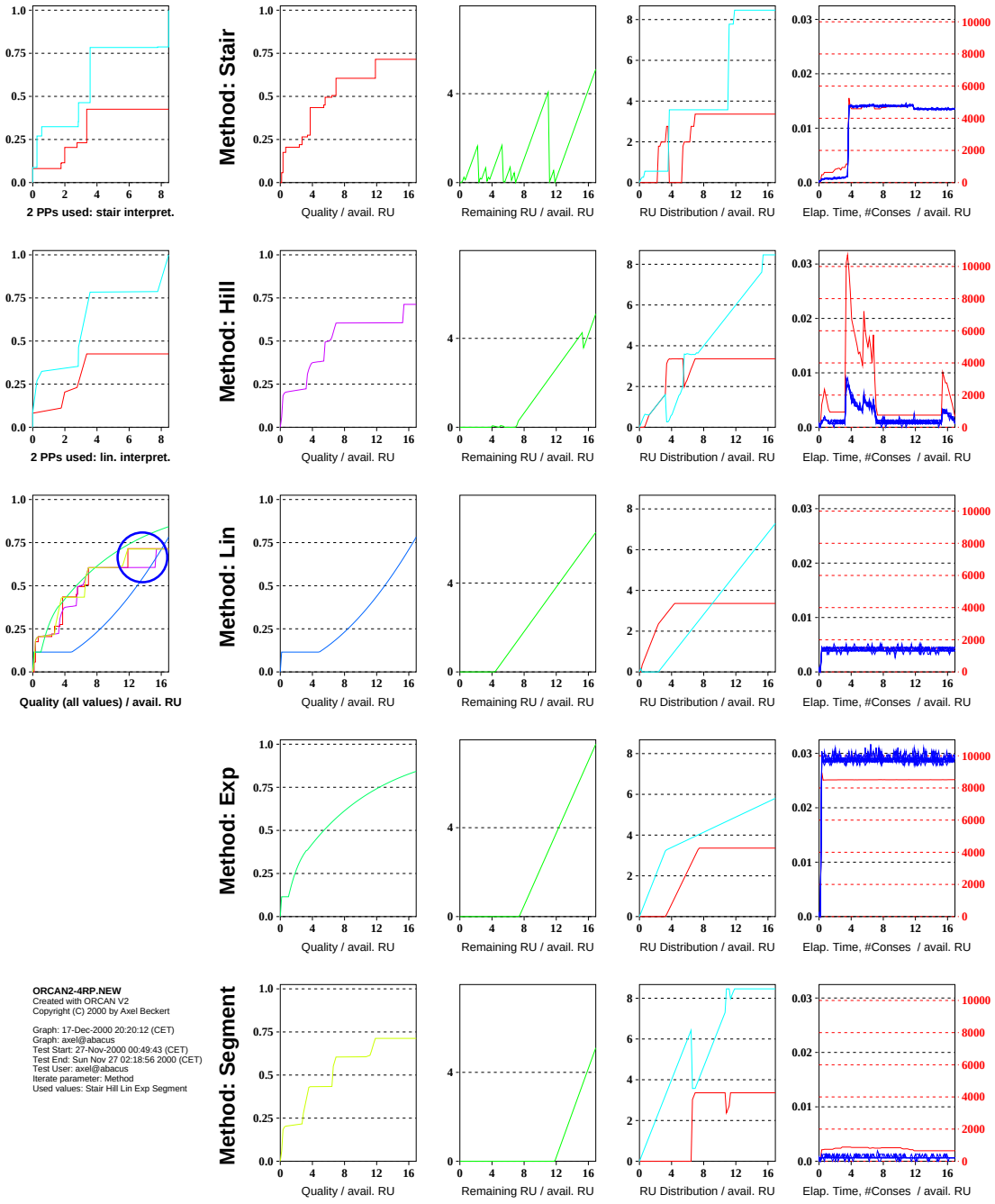

Abbildung A.2.: Laufzeittest mit 2 Performanzprofilen und Iteration über alle Kompilierungsmethoden

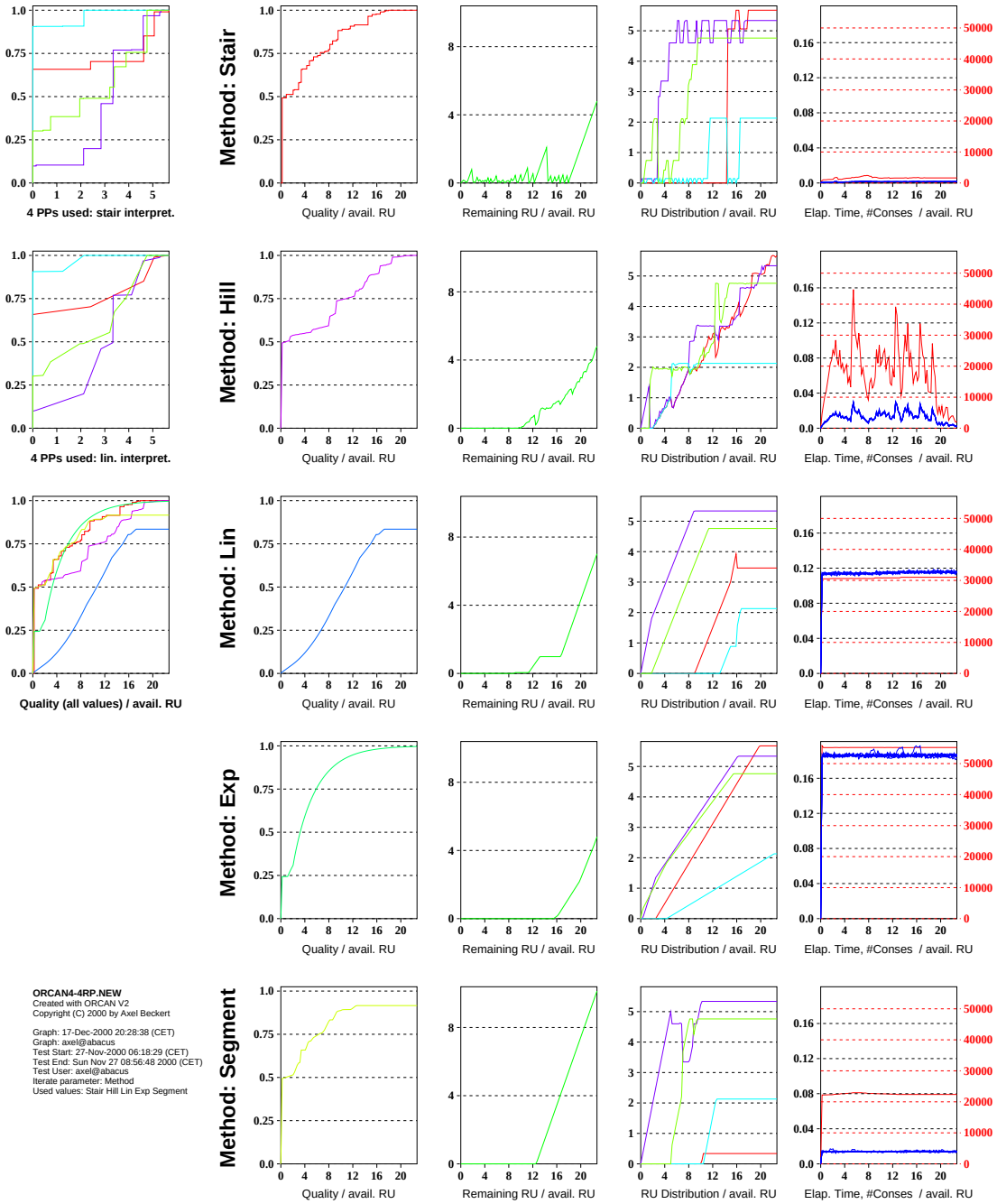

Abbildung A.3.: Laufzeittest mit 4 Performanzprofilen und Iteration über alle Kompilierungsmethoden

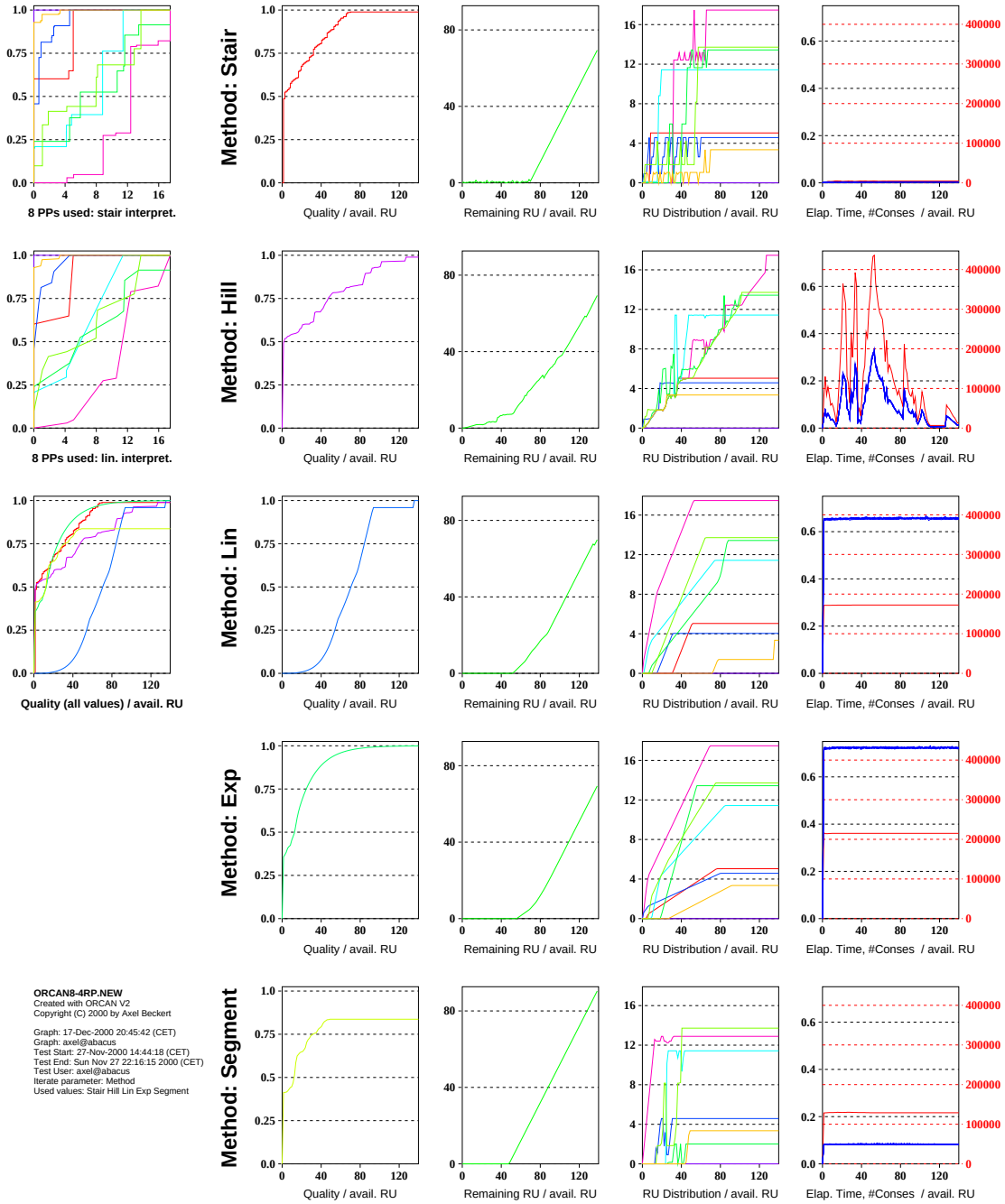

Abbildung A.4.: Laufzeittest mit 8 Performanzprofilen und Iteration über alle Kompilierungsmethoden

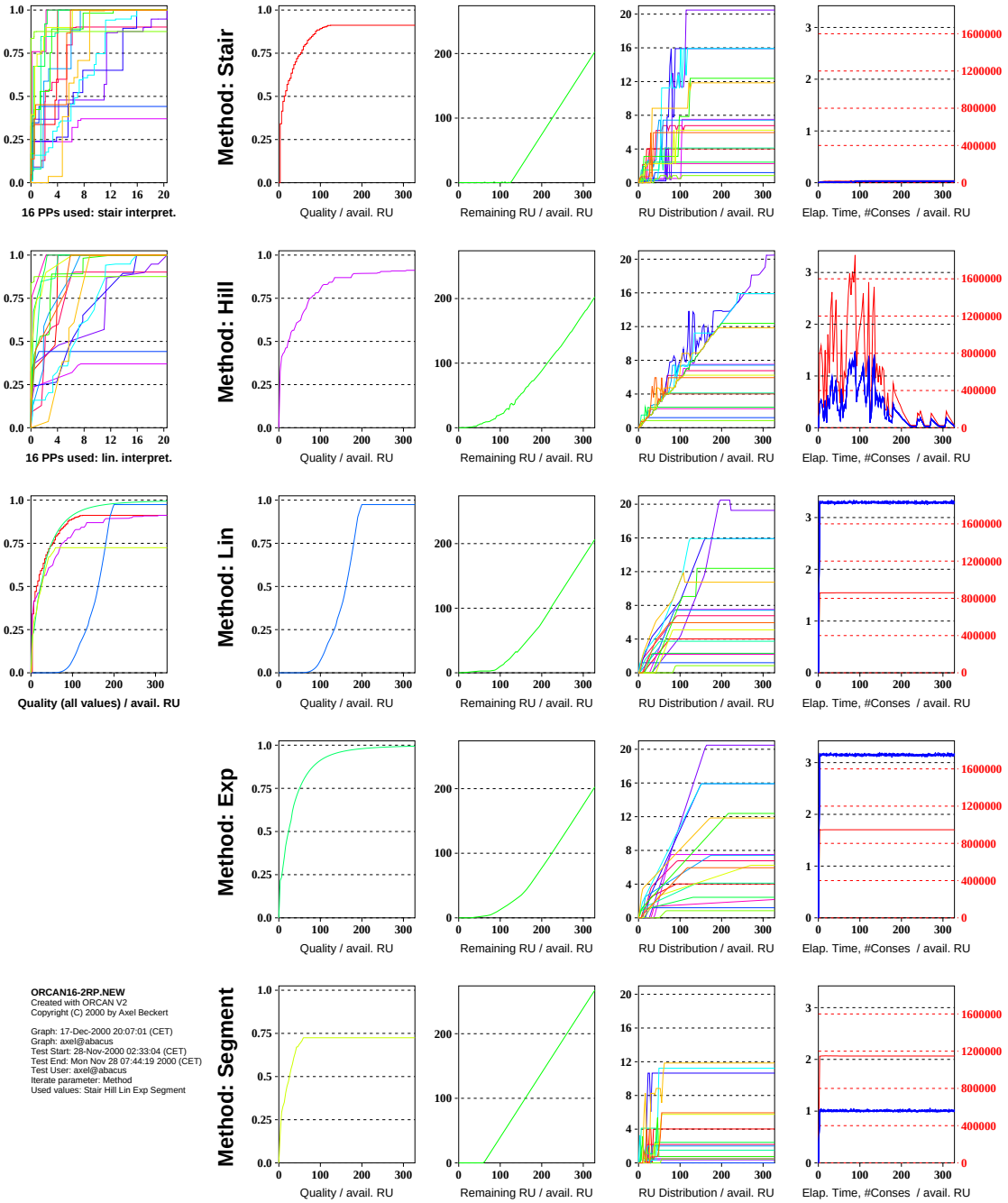

Abbildung A.5.: Laufzeittest mit 16 Performanzprofilen und Iteration über alle Kompilierungsmethoden

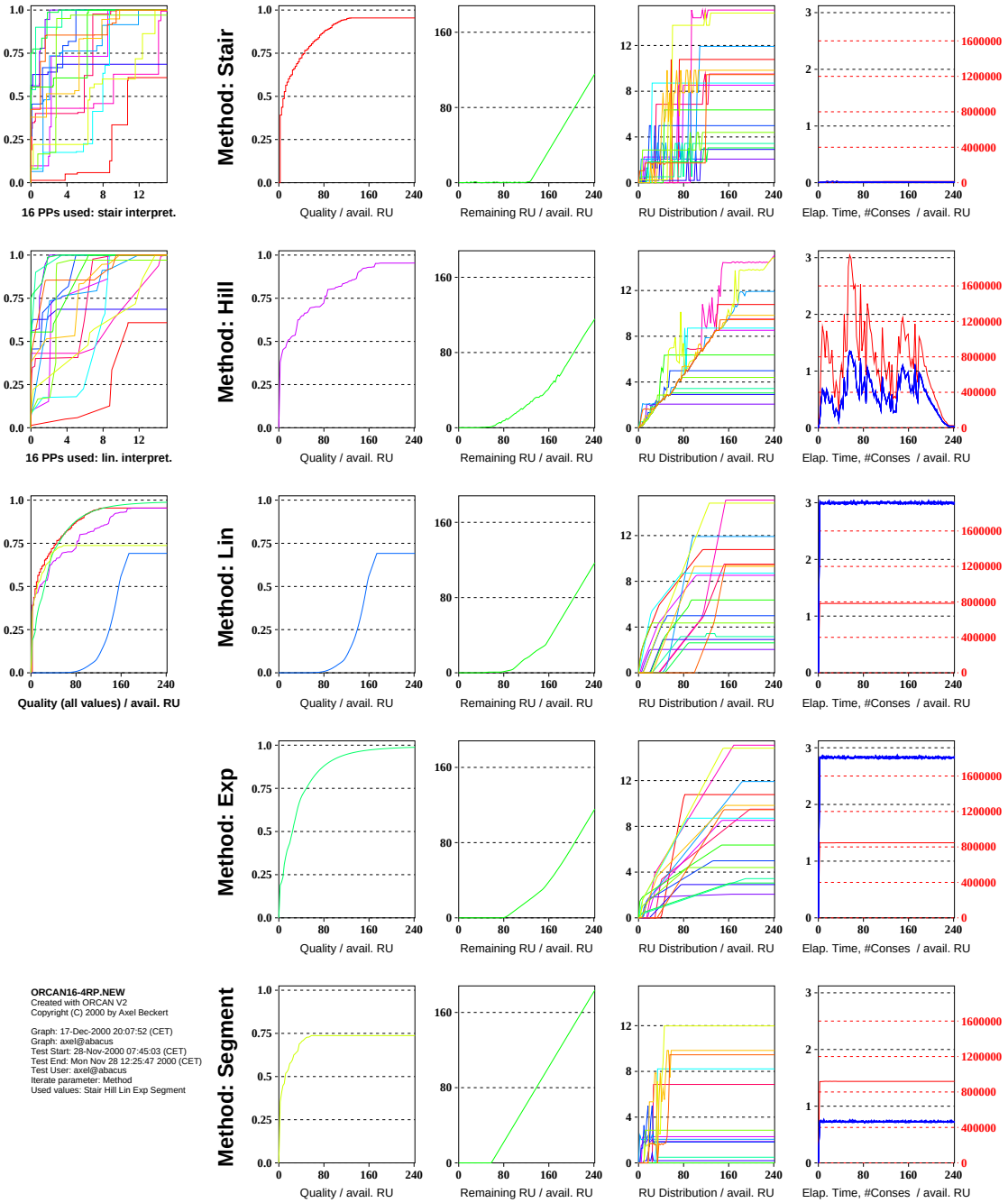

Abbildung A.6.: Laufzeittest mit 16 Performanzprofilen und Iteration über alle Kompilierungsmethoden

A.1.2. Performanzprofile mit unterschiedlichen Stützpunktzahlen

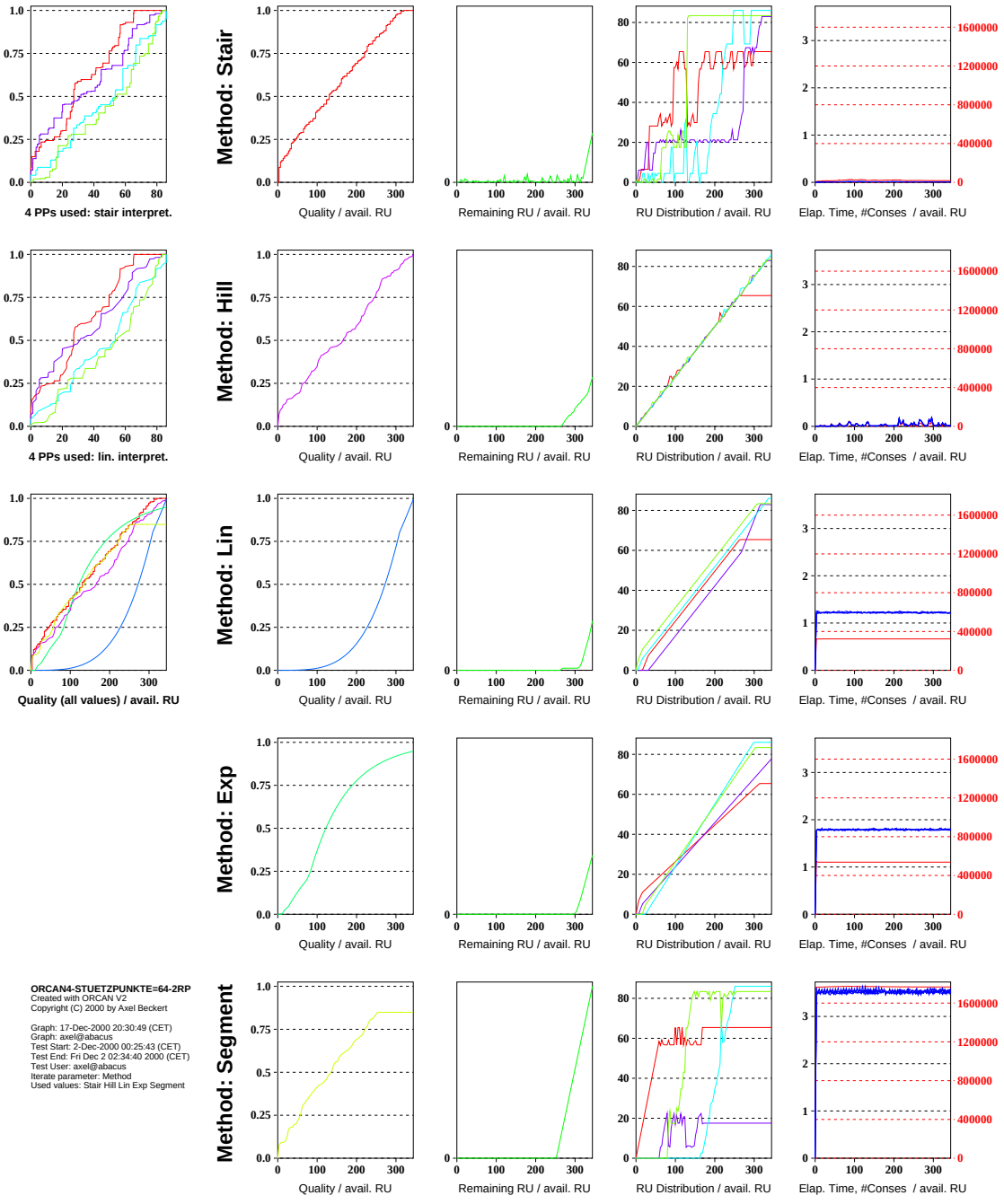

Abbildung A.7.: Laufzeittest mit 4 Performanzprofilen mit je 64 Stützpunkten und Iteration über alle Kompilierungsmethoden

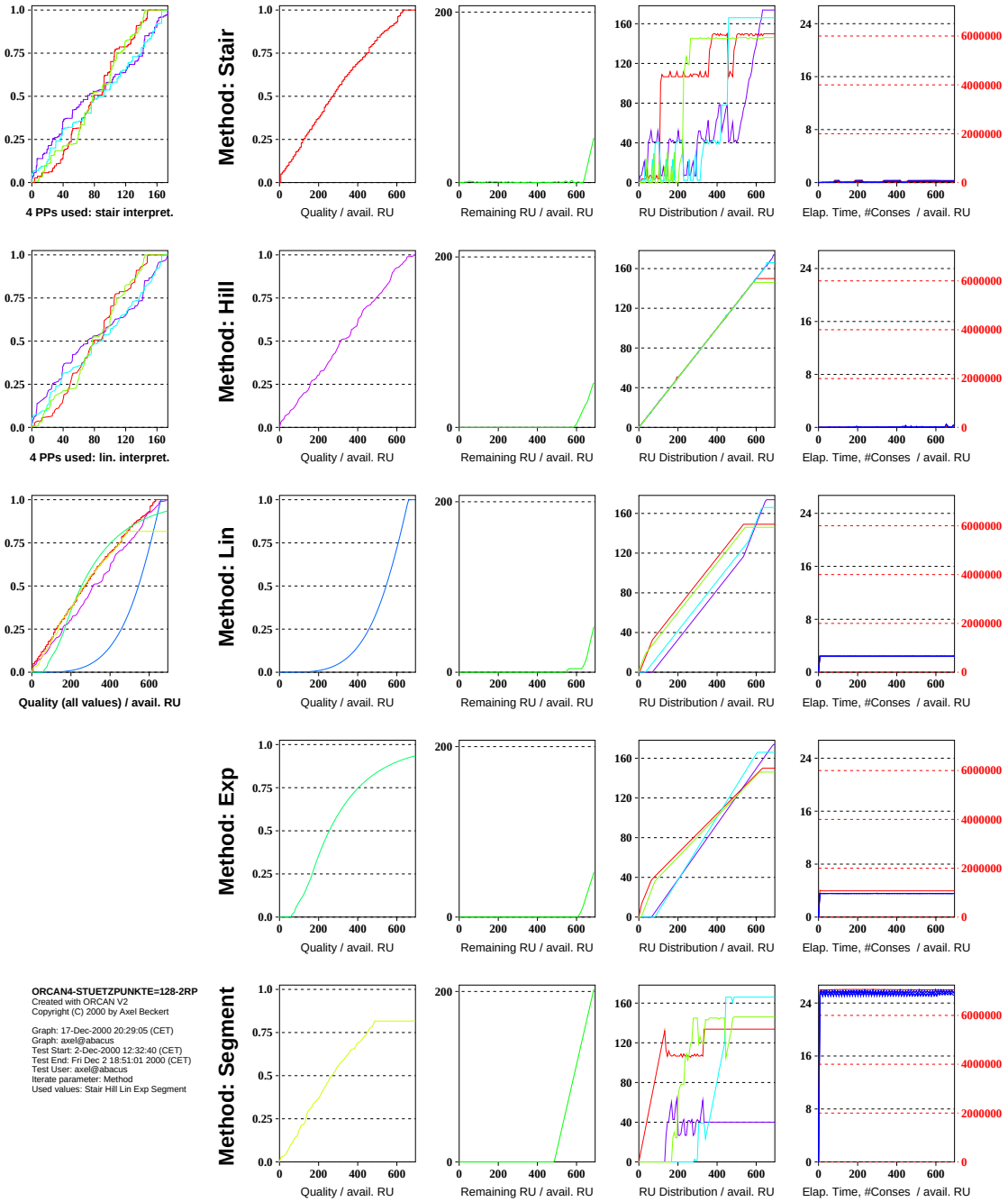

Abbildung A.8.: Laufzeittest mit 4 Performanzprofilen mit je 128 Stützpunkten und Iteration über alle Kompilierungsmethoden

A.2. Vergleich verschiedener Parameterwerte bei einzelnen Kompilierungsmethoden

A.2.1. Parameter der Hillclimbing-Methode

A.2.1.1. Parameter `:combine` der Hillclimbing-Methode

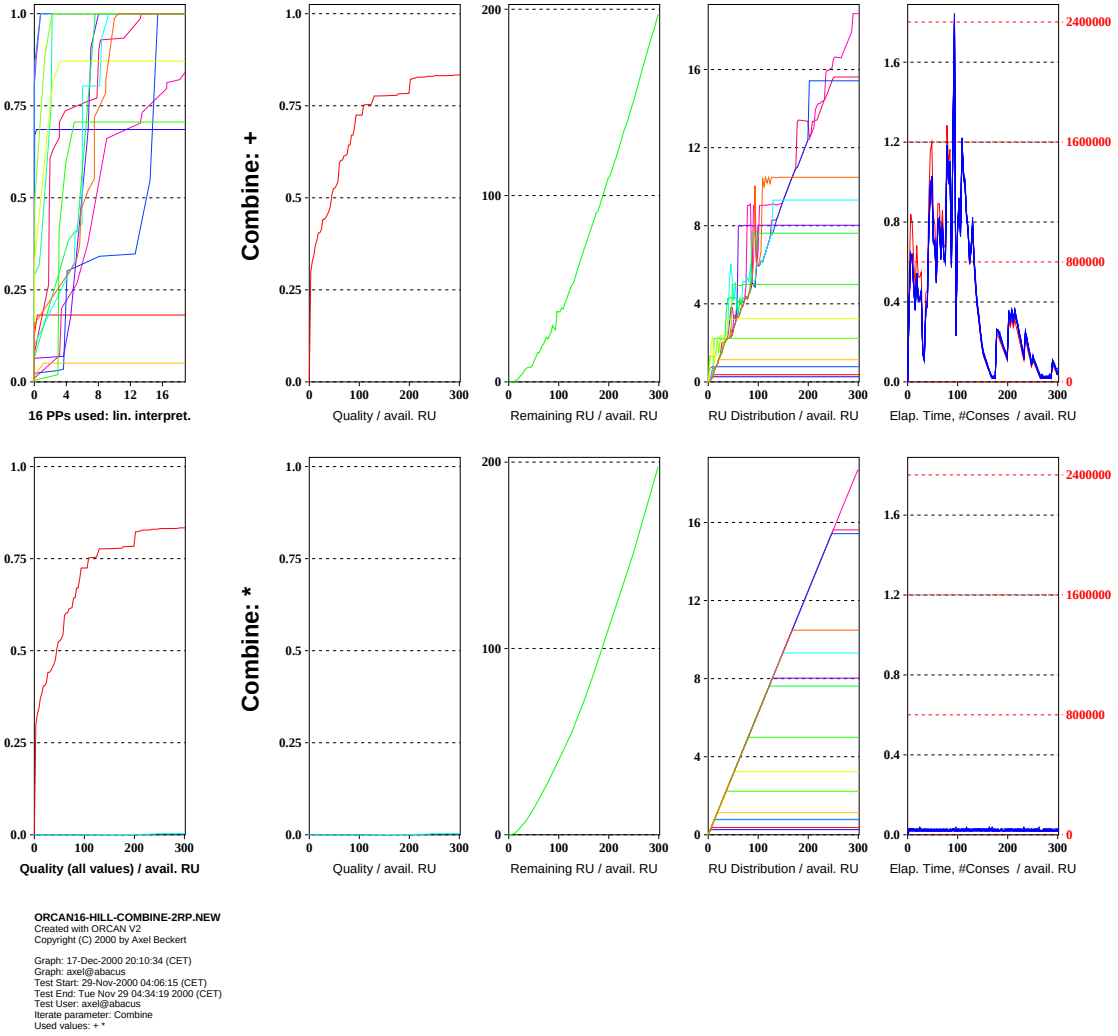

Abbildung A.9.: Laufzeittest mit 16 Performanzprofilen und Iteration des Hillclimbing-Algorithmus-Parameters `:combine` über die Werte `#'*` und `#'+`

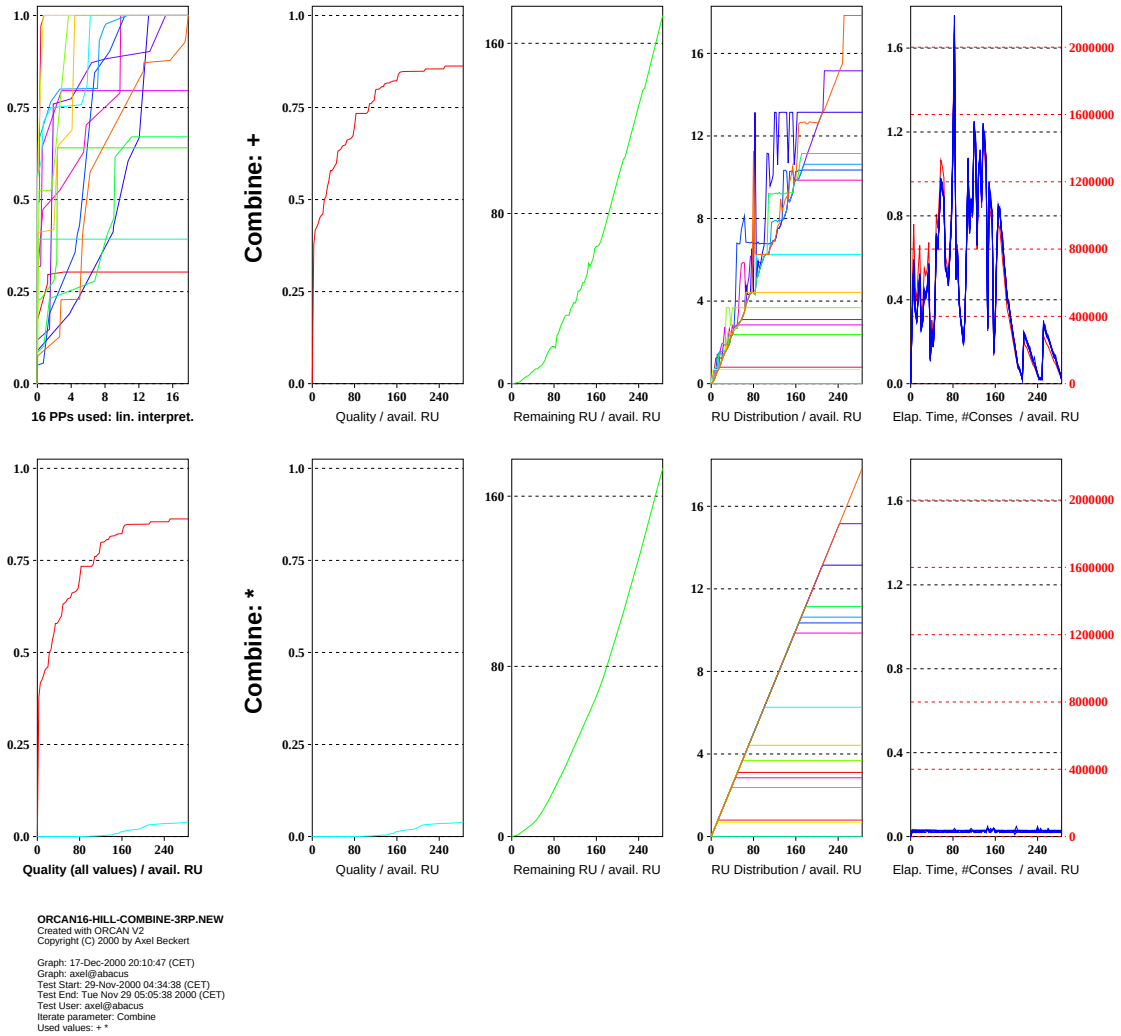

Abbildung A.10.: Laufzeittest mit 16 Performanzprofilen und Iteration des Hill-climbing-Algorithmus-Parameters `:combine` über die Werte `#'*` und `#'+`

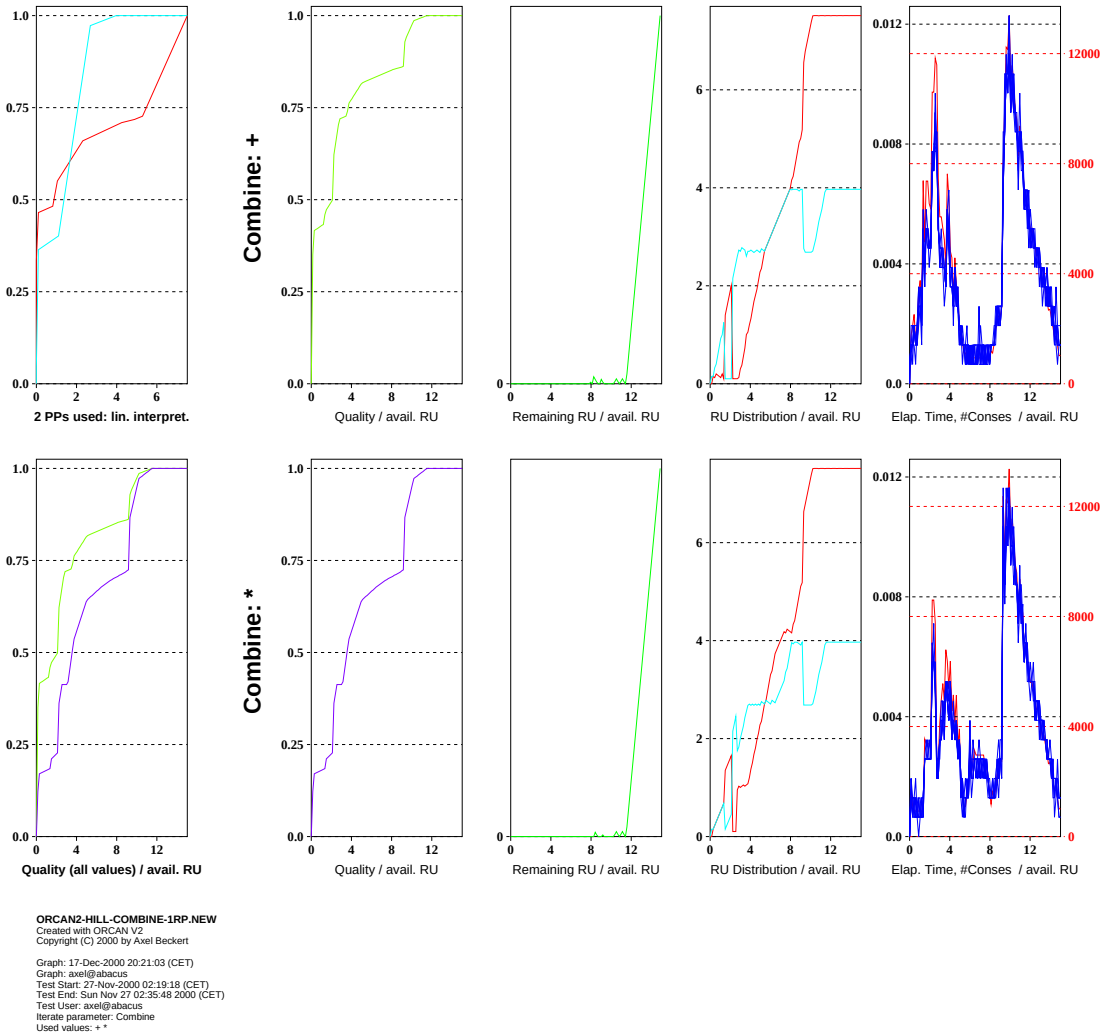

Abbildung A.11.: Laufzeittest mit 2 Performanzprofilen und Iteration des Hill-climbing-Algorithmus-Parameters `:combine` über die Werte `#'*` und `#'+`

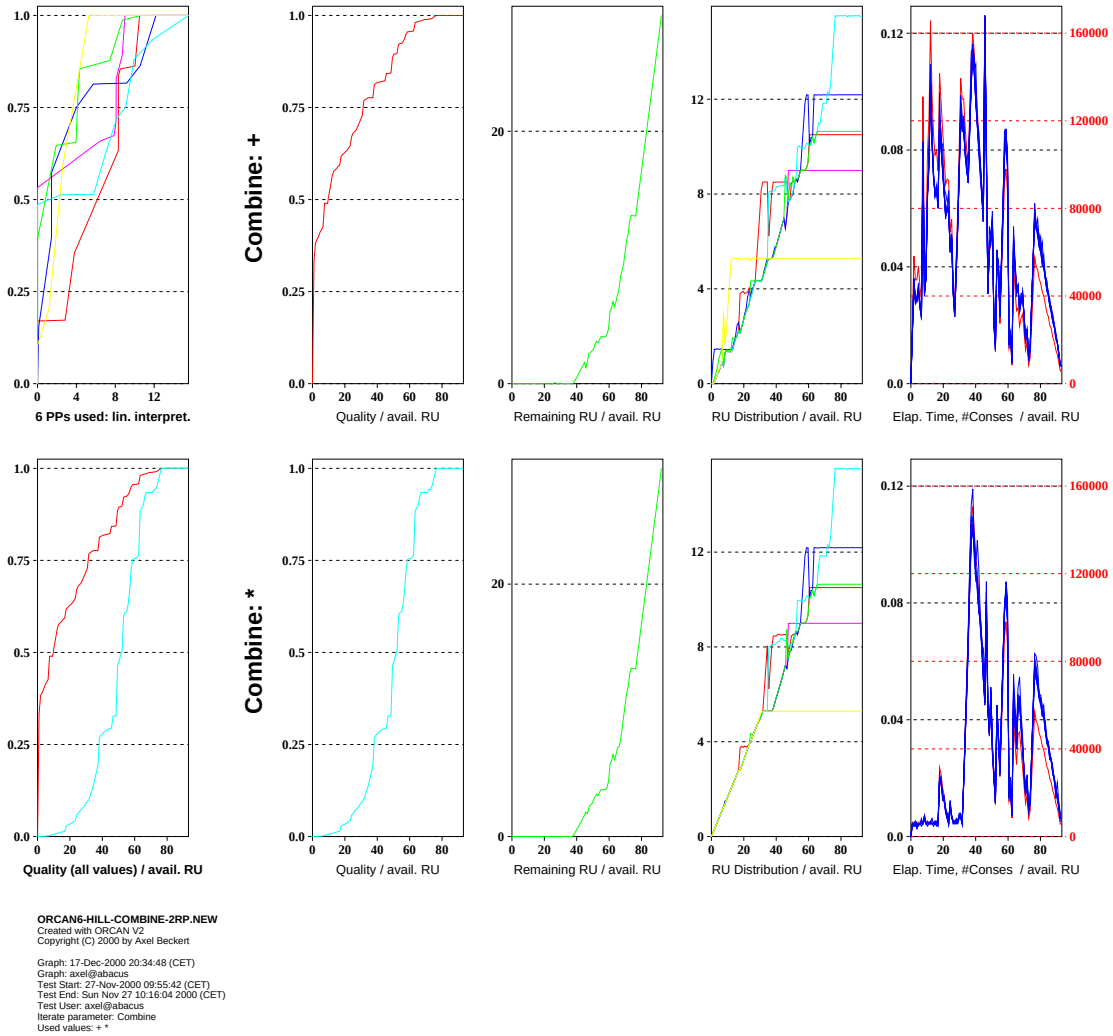

Abbildung A.12.: Laufzeittest mit 6 Performanzprofilen und Iteration des Hill-climbing-Algorithmus-Parameters `:combine` über die Werte `#'*` und `#'+`

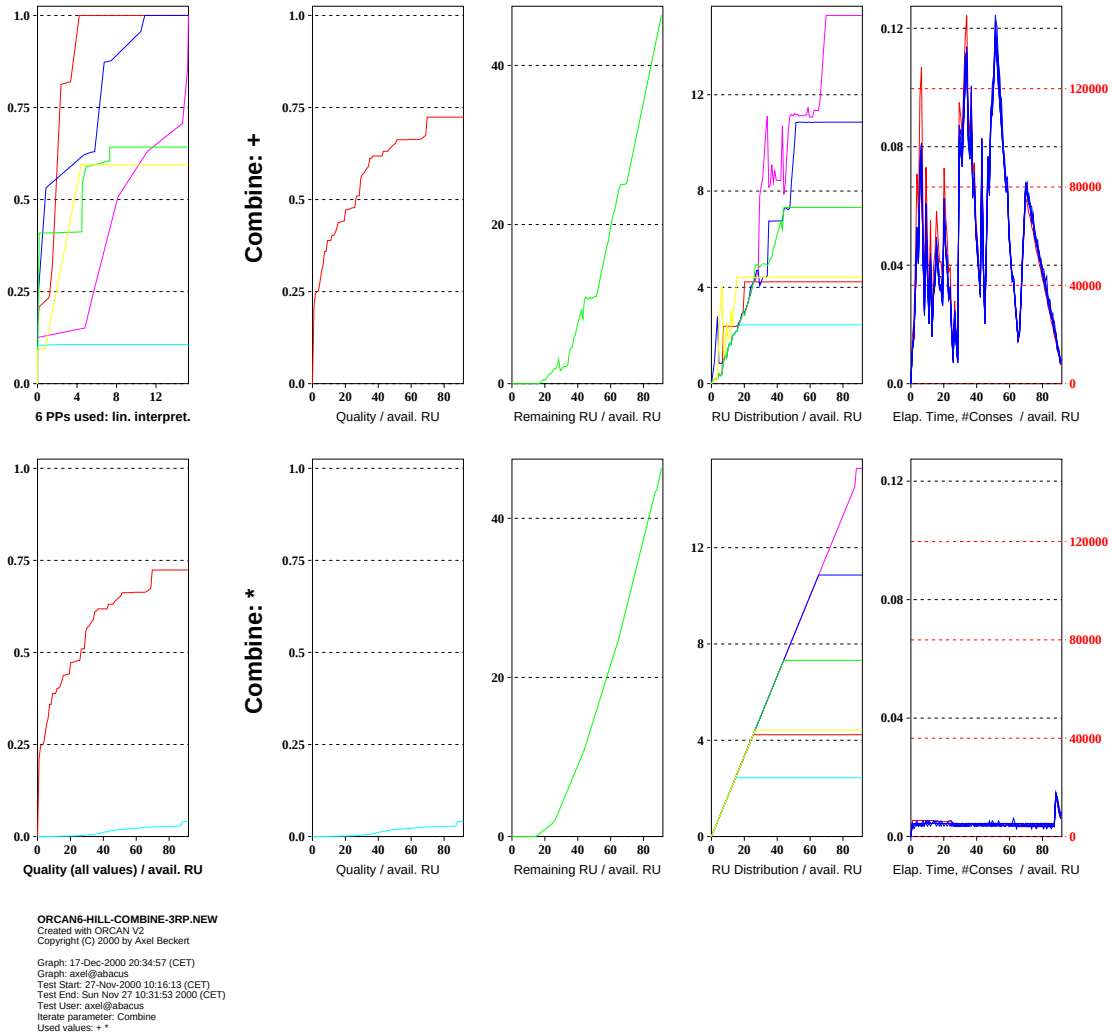

Abbildung A.13.: Laufzeittest mit 6 Performanzprofilen und Iteration des Hill-climbing-Algorithmus-Parameters `:combine` über die Werte `#'*` und `#'+`

A.2.1.2. Parameter `:logical-operator` der Hillclimbing-Methode

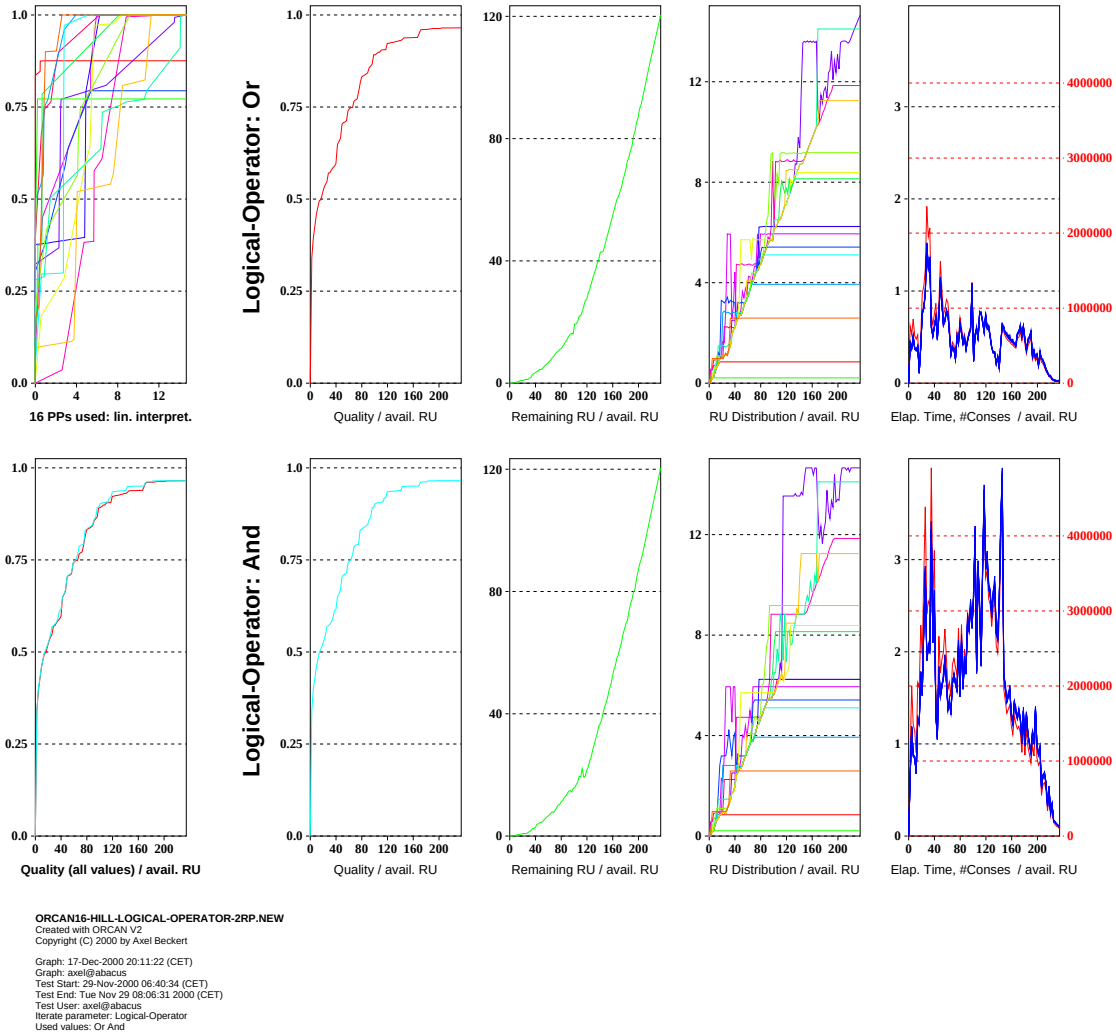

Abbildung A.14.: Laufzeittest mit 16 Performanzprofilen und Iteration des Parameters `:logical-operator` über die Werte `#'and` und `#'or`

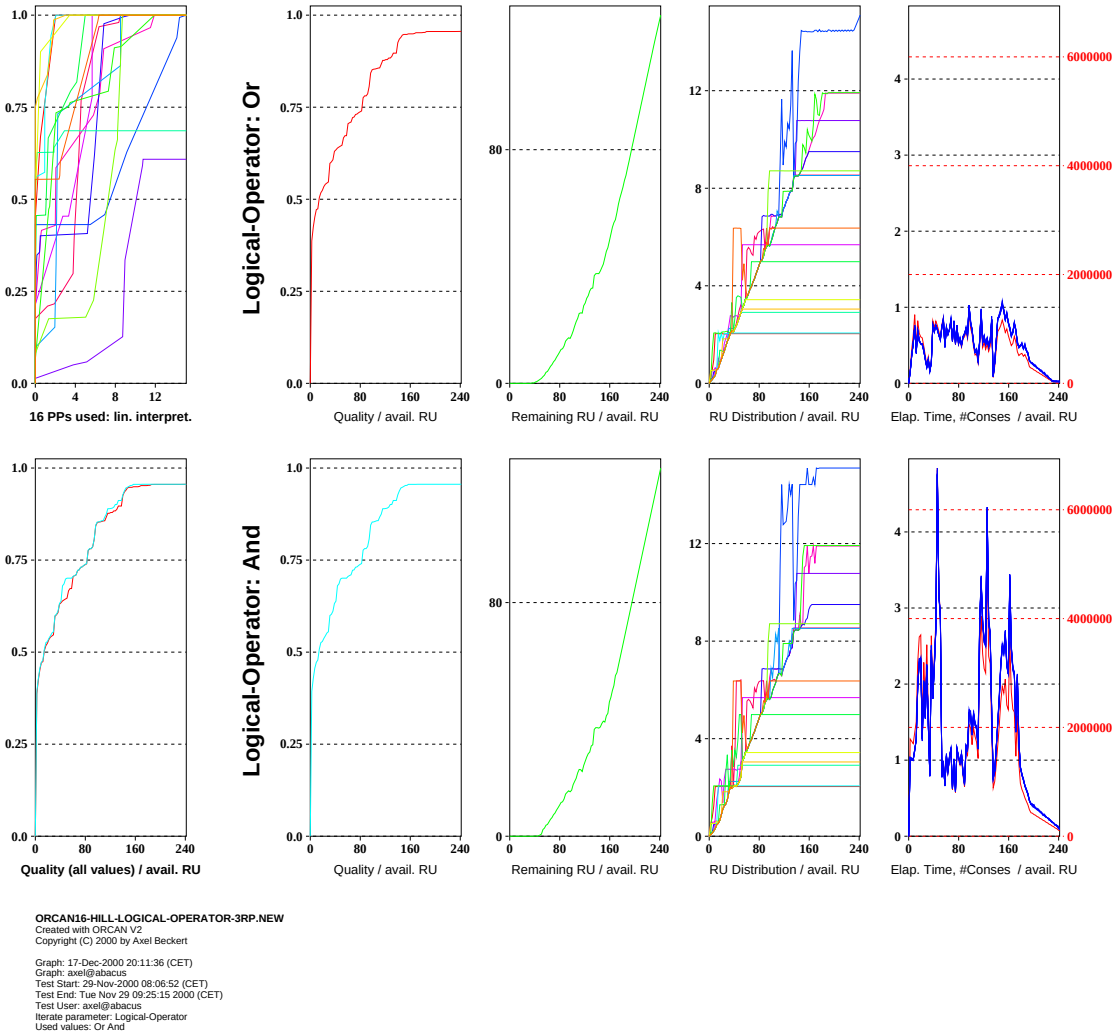

Abbildung A.15.: Laufzeittest mit 16 Performanzprofilen und Iteration des Parameters `:logical-operator` über die Werte `#'and` und `#'or`

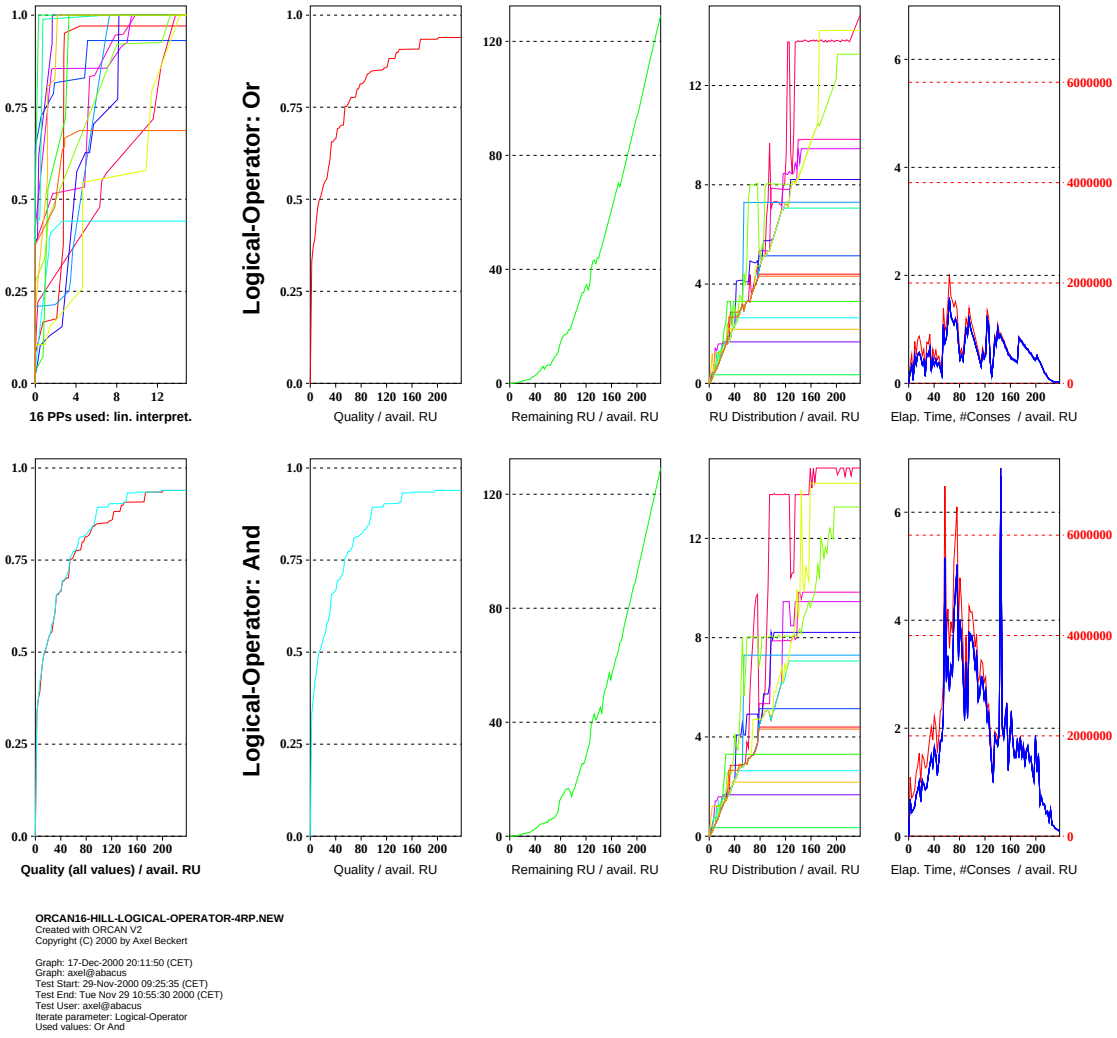

Abbildung A.16.: Laufzeittest mit 16 Performanzprofilen und Iteration des Parameters `:logical-operator` über die Werte `#'and` und `#'or`

A.2.1.3. Parameter `:tolerance` der Hillclimbing-Methode

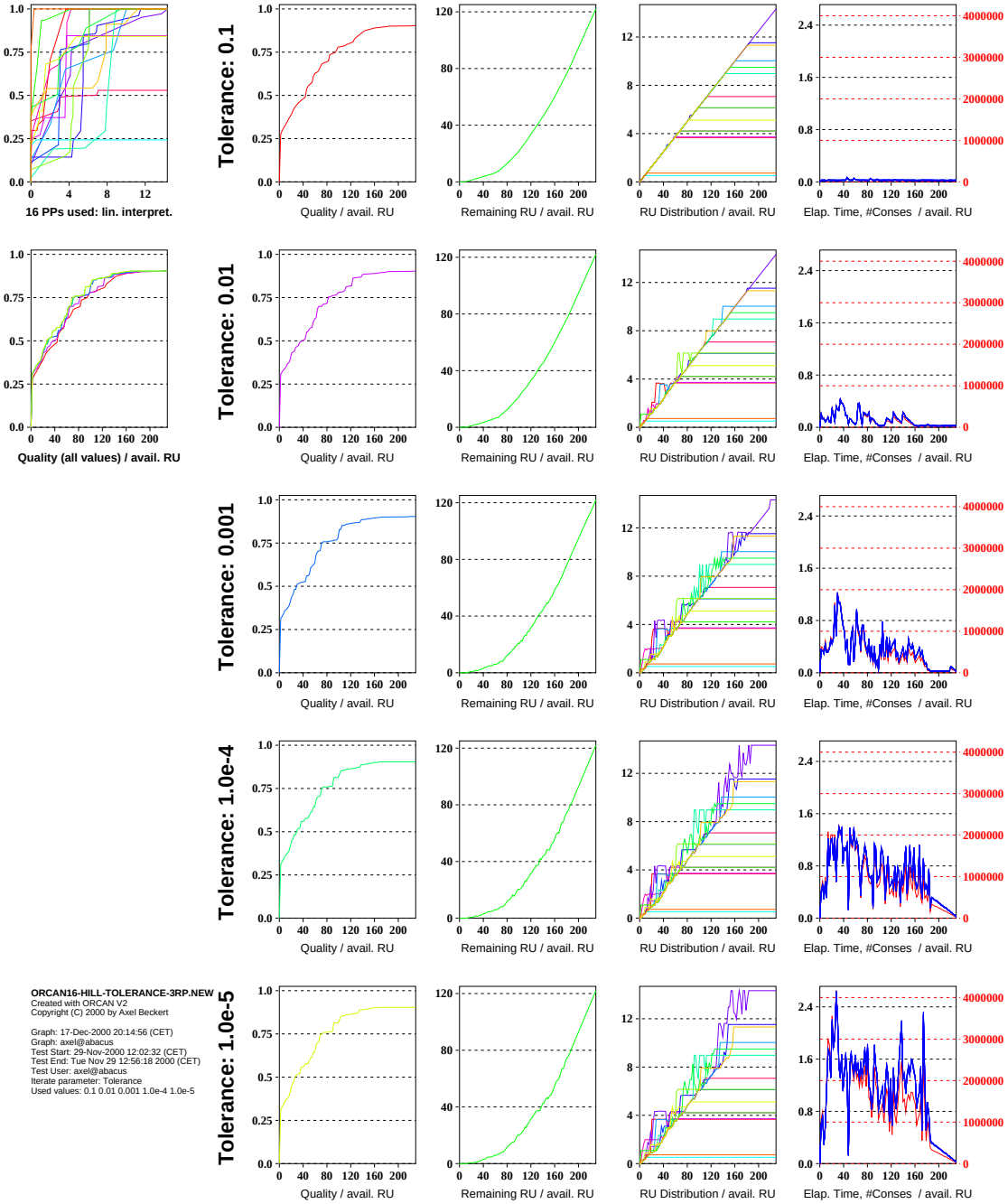

Abbildung A.17.: Laufzeittest mit 16 Performanzprofilen und Iteration des Hillclimbing-Algorithmus-Parameters `:tolerance` über die Werte 10^{-n} mit $n \in \{1, \ldots, 5\}$

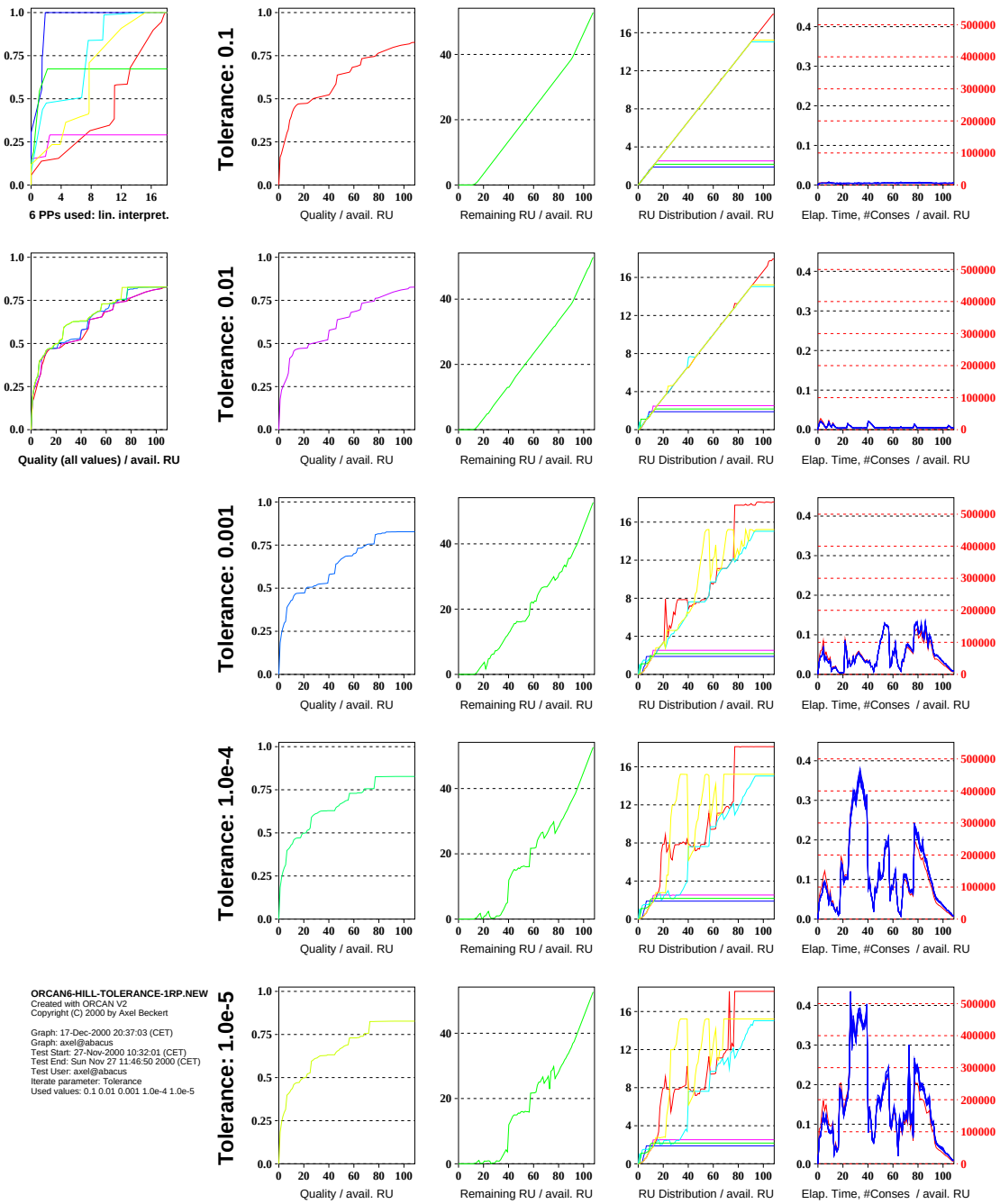

Abbildung A.18.: Laufzeittest mit 6 Performanzprofilen und Iteration des Hill-climbing-Algorithmus-Parameters `:tolerance` über die Werte 10^{-n} mit $n \in \{1, \ldots, 5\}$

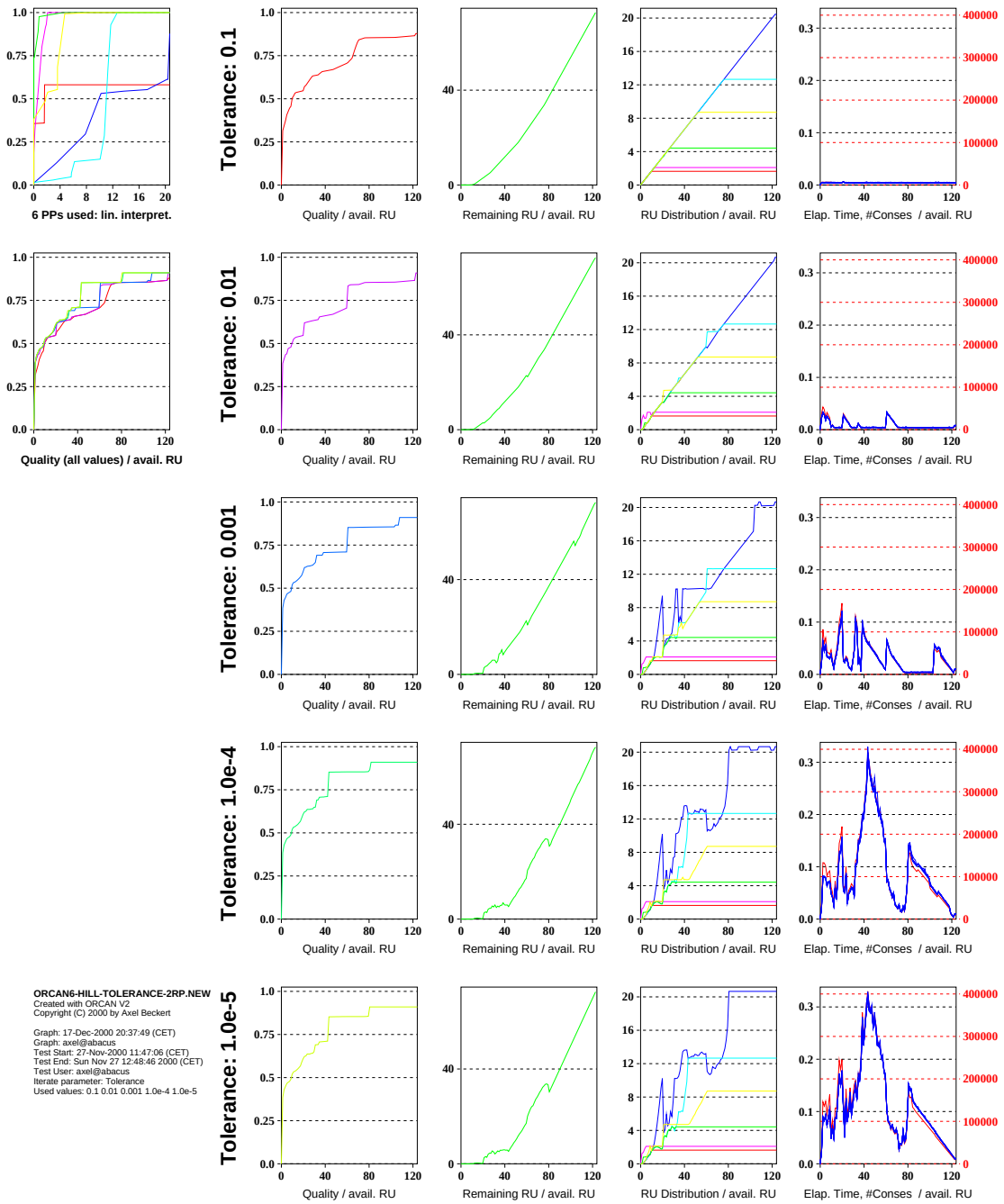

Abbildung A.19.: Laufzeittest mit 6 Performanzprofilen und Iteration des Hill-climbing-Algorithmus-Parameters `:tolerance` über die Werte 10^{-n} mit $n \in \{1, \ldots, 5\}$

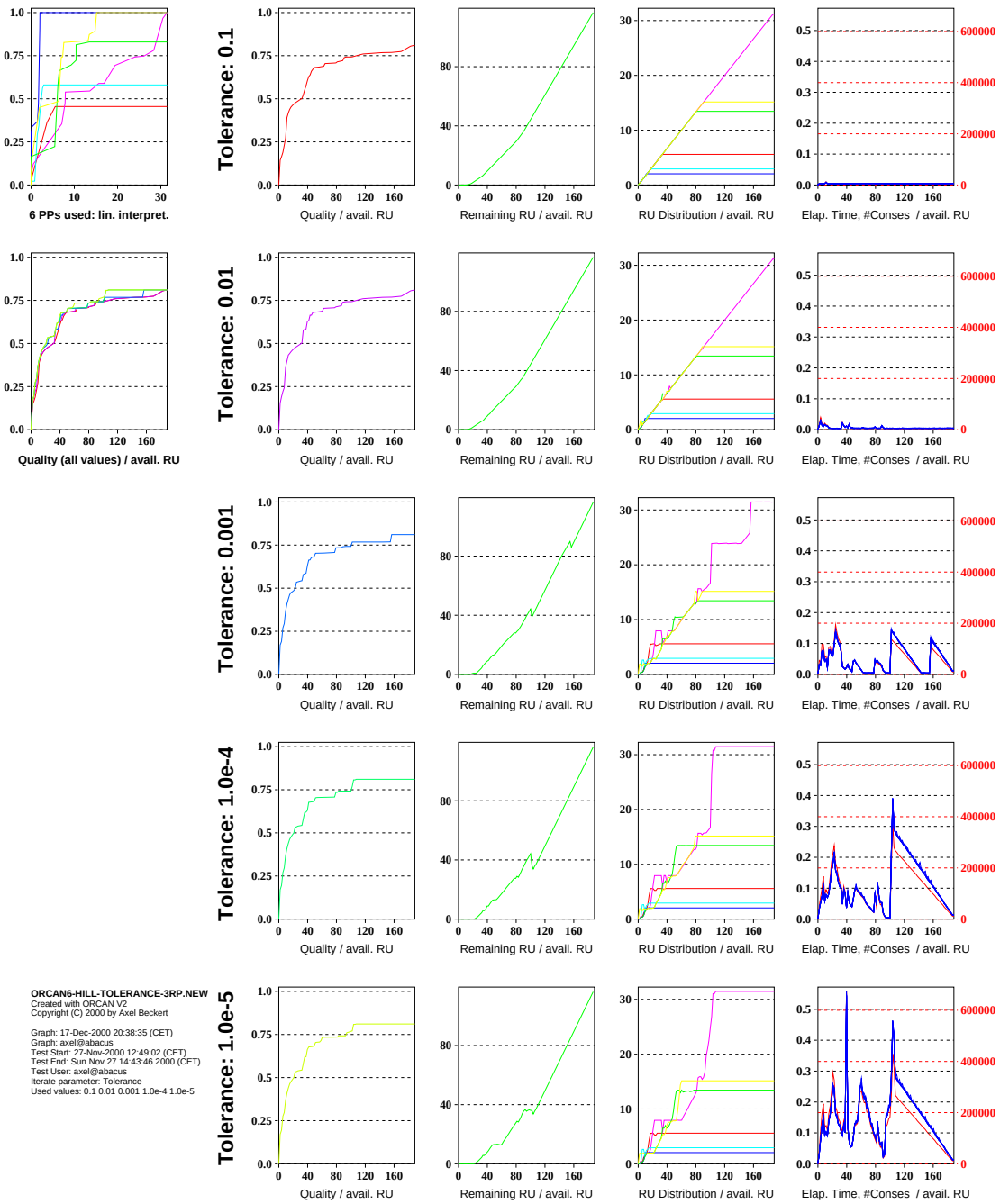

Abbildung A.20.: Laufzeittest mit 6 Performanzprofilen und Iteration des Hill-climbing-Algorithmus-Parameters `:tolerance` über die Werte 10^{-n} mit $n \in \{1, \ldots, 5\}$

A.2.1.4. Parameter `:always-test-dir` der Hillclimbing-Methode

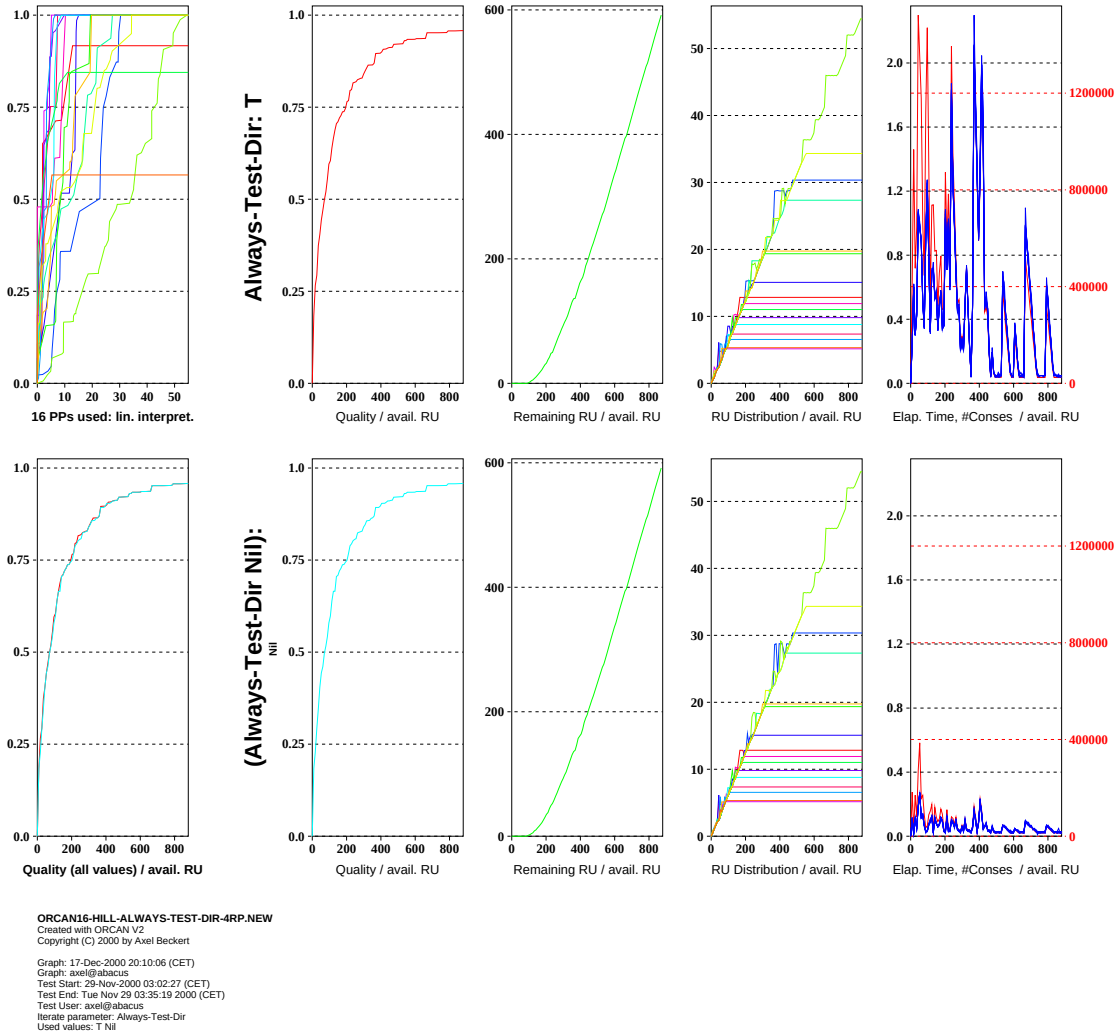

Abbildung A.21.: Laufzeittest mit 16 Performanzprofilen und Iteration des Parameters `:always-test-dir` über die Werte `t` und `nil`

A.2.2. Parameter der Treppenstufen-Methode

A.2.2.1. Parameter `:type` der Treppenstufen-Methode mit $\alpha = 1$ und $\beta = 0$

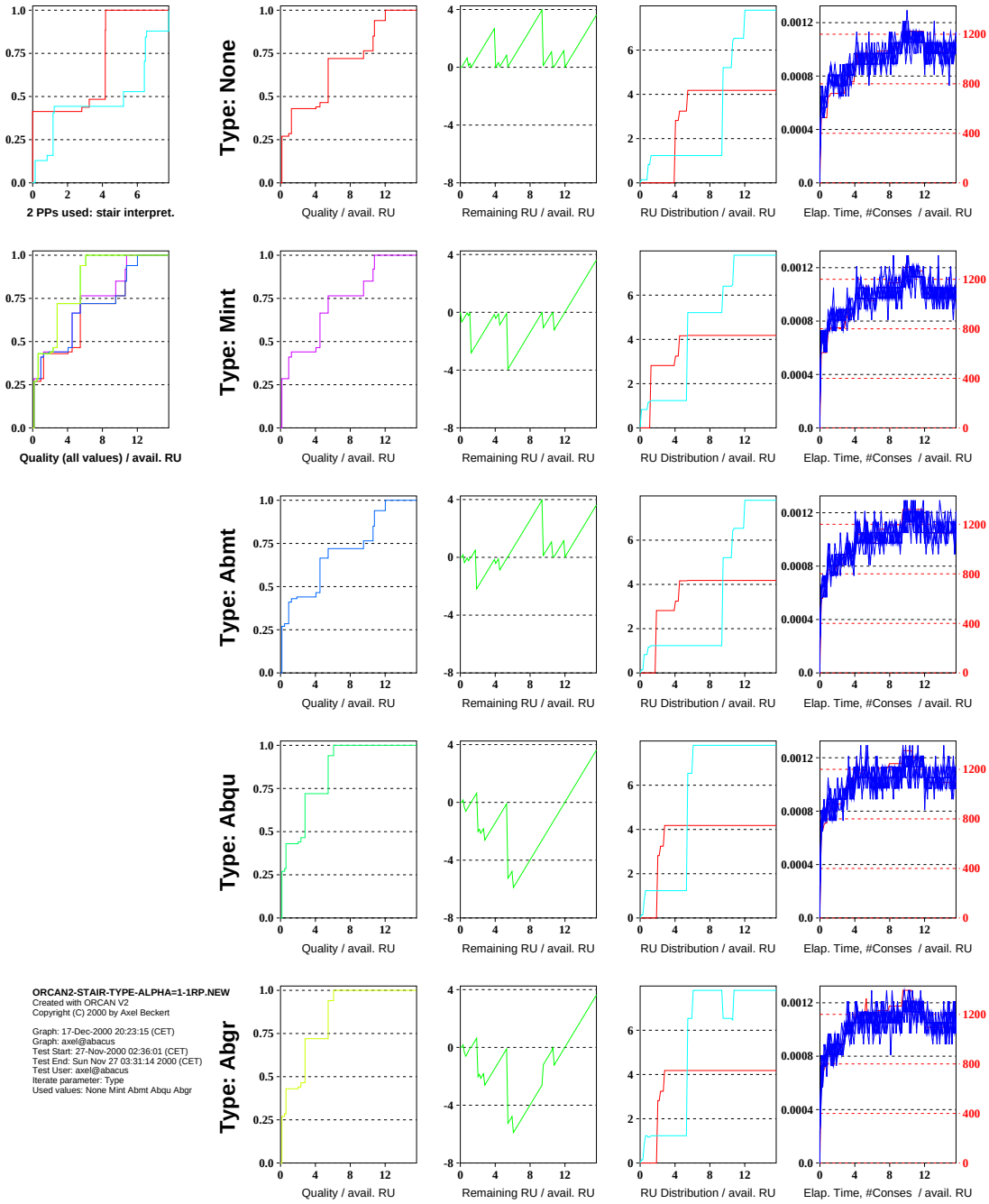

Abbildung A.22.: Laufzeittest mit 2 Performanzprofilen und Iteration des Treppenstufen-Methode-Parameters `:type` (bei $\alpha = 1$ und $\beta = 0$) über sämtliche möglichen Werte

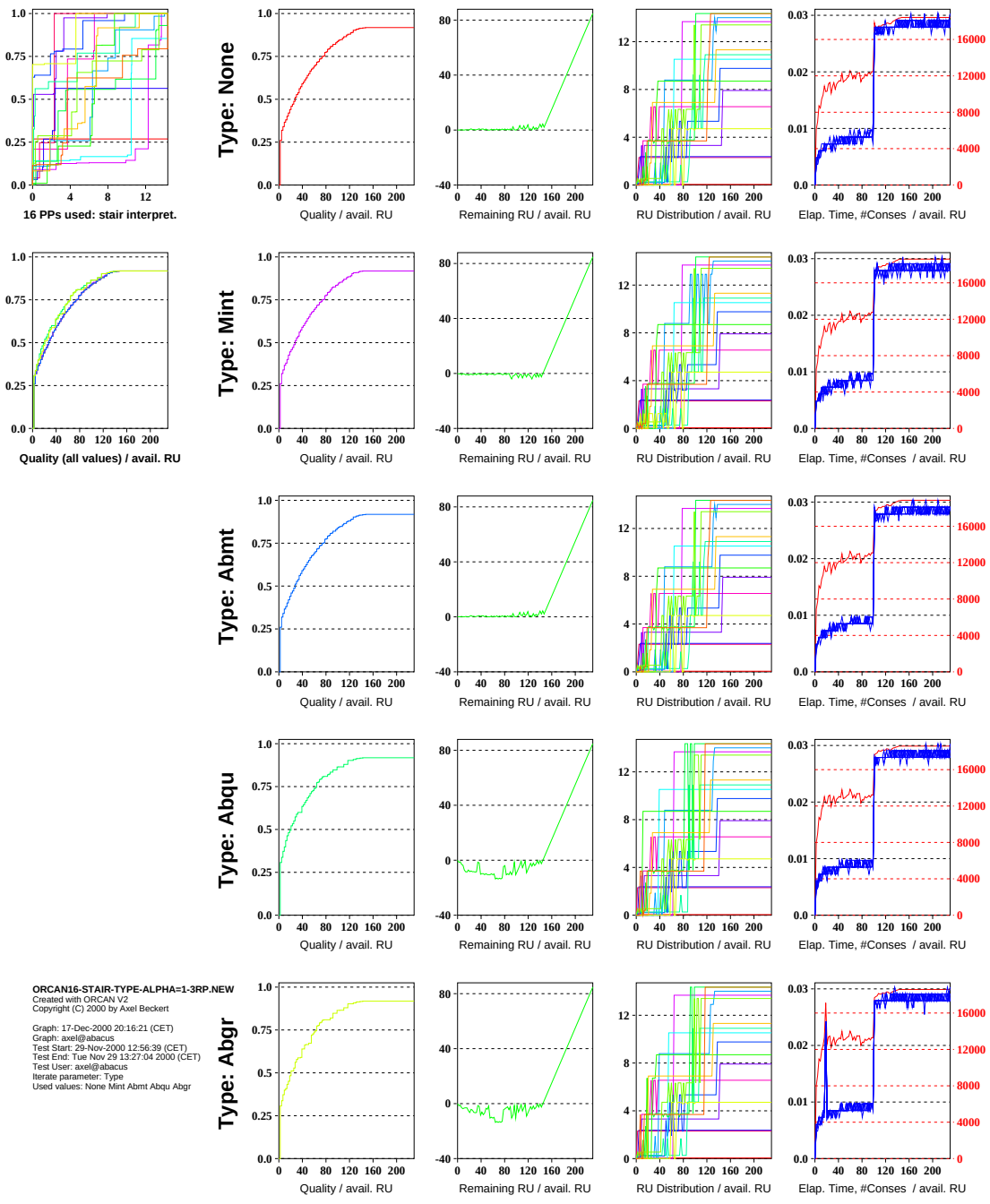

Abbildung A.23.: Laufzeittest mit 16 Performanzprofilen und Iteration des Treppenstufen-Methode-Parameters `:type` (bei $\alpha = 1$ und $\beta = 0$) über sämtliche möglichen Werte

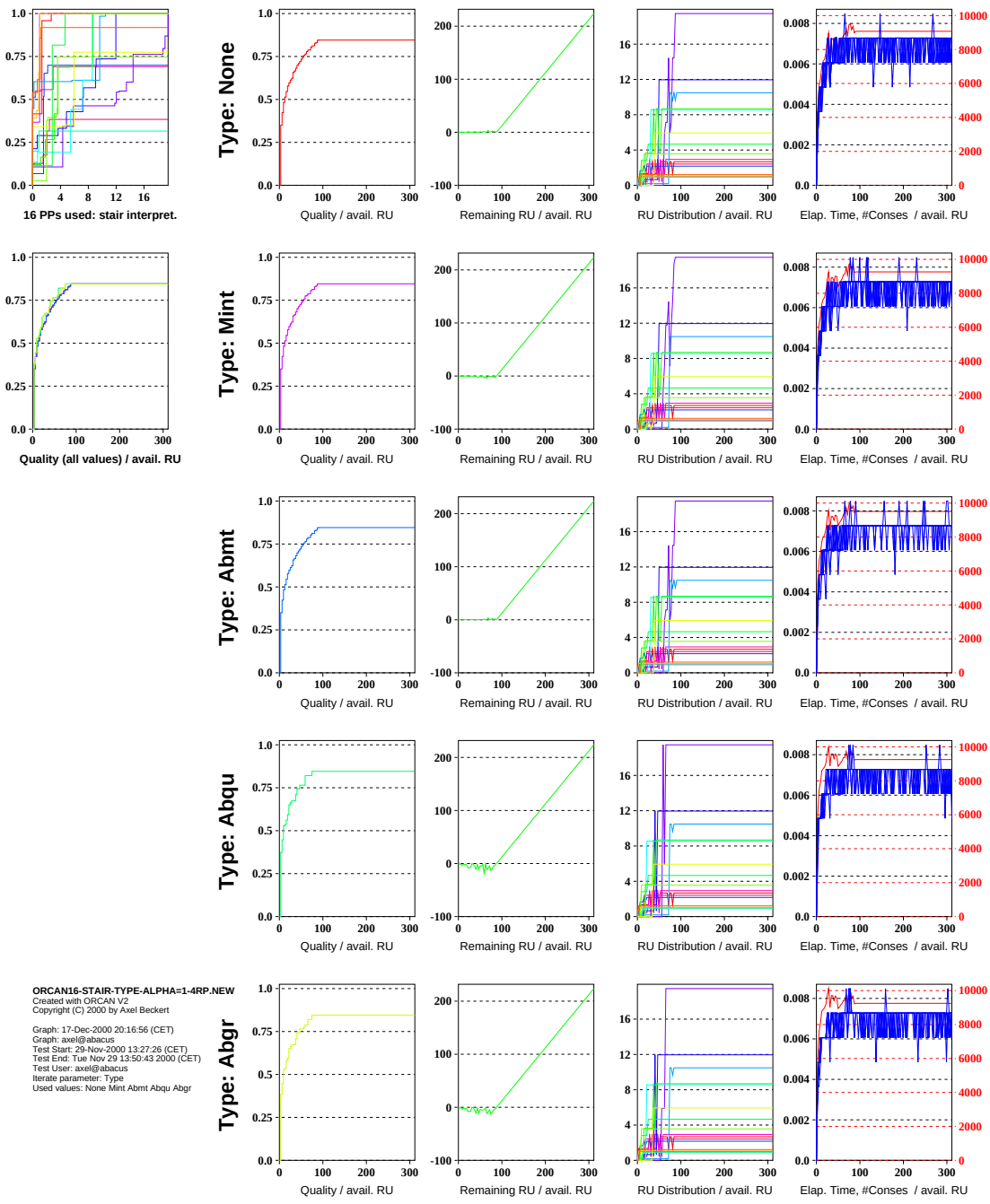

Abbildung A.24.: Laufzeittest mit 16 Performanzprofilen und Iteration des Treppenstufen-Methode-Parameters `:type` (bei $\alpha = 1$ und $\beta = 0$) über sämtliche möglichen Werte

A.2.2.2. Parameter `:type` der Treppenstufen-Methode mit $\alpha = 0$ und $\beta = 1$

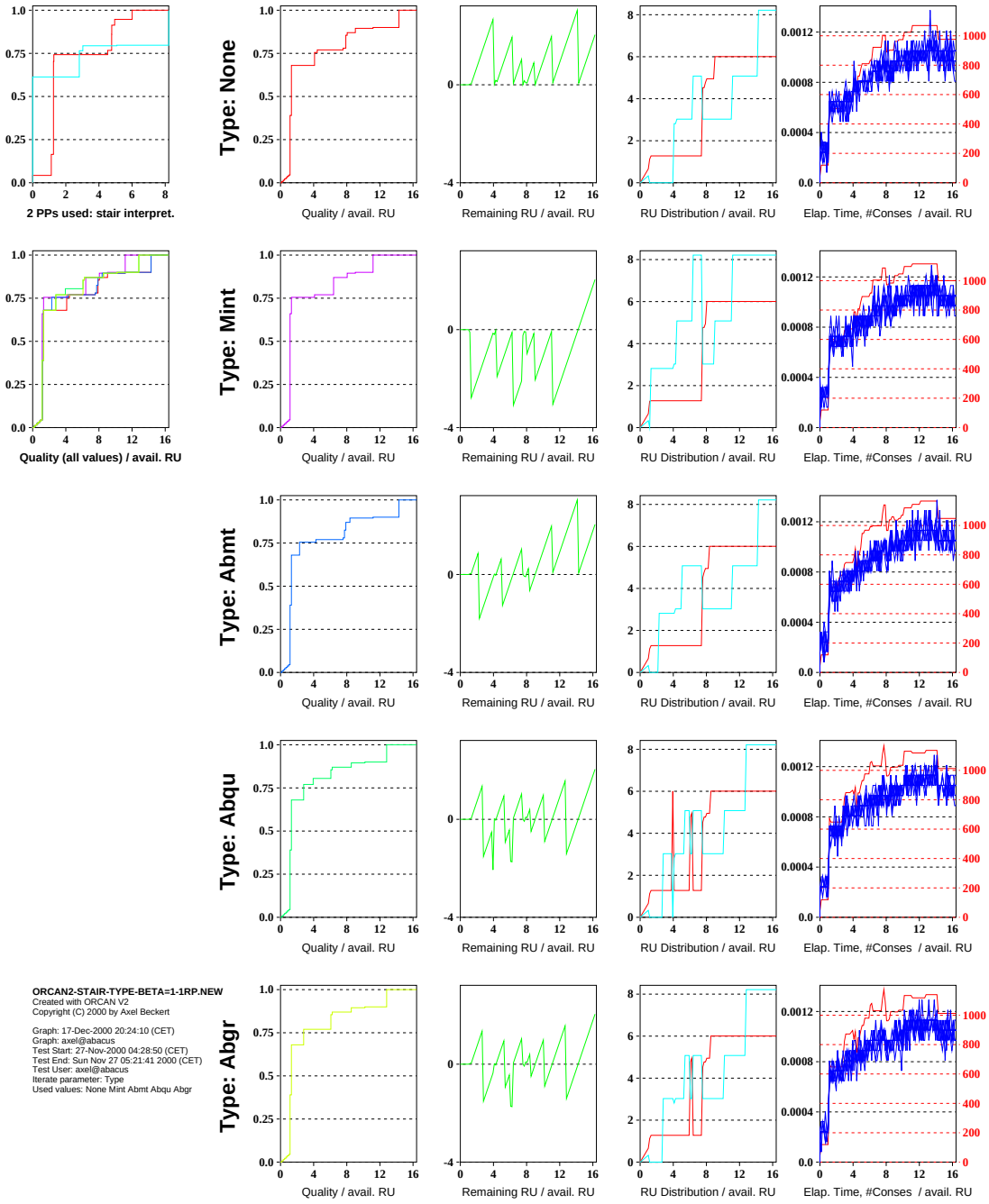

Abbildung A.25.: Laufzeittest mit 2 Performanzprofilen und Iteration des Treppenstufen-Methode-Parameters `:type` (bei $\alpha = 0$ und $\beta = 1$) über sämtliche möglichen Werte

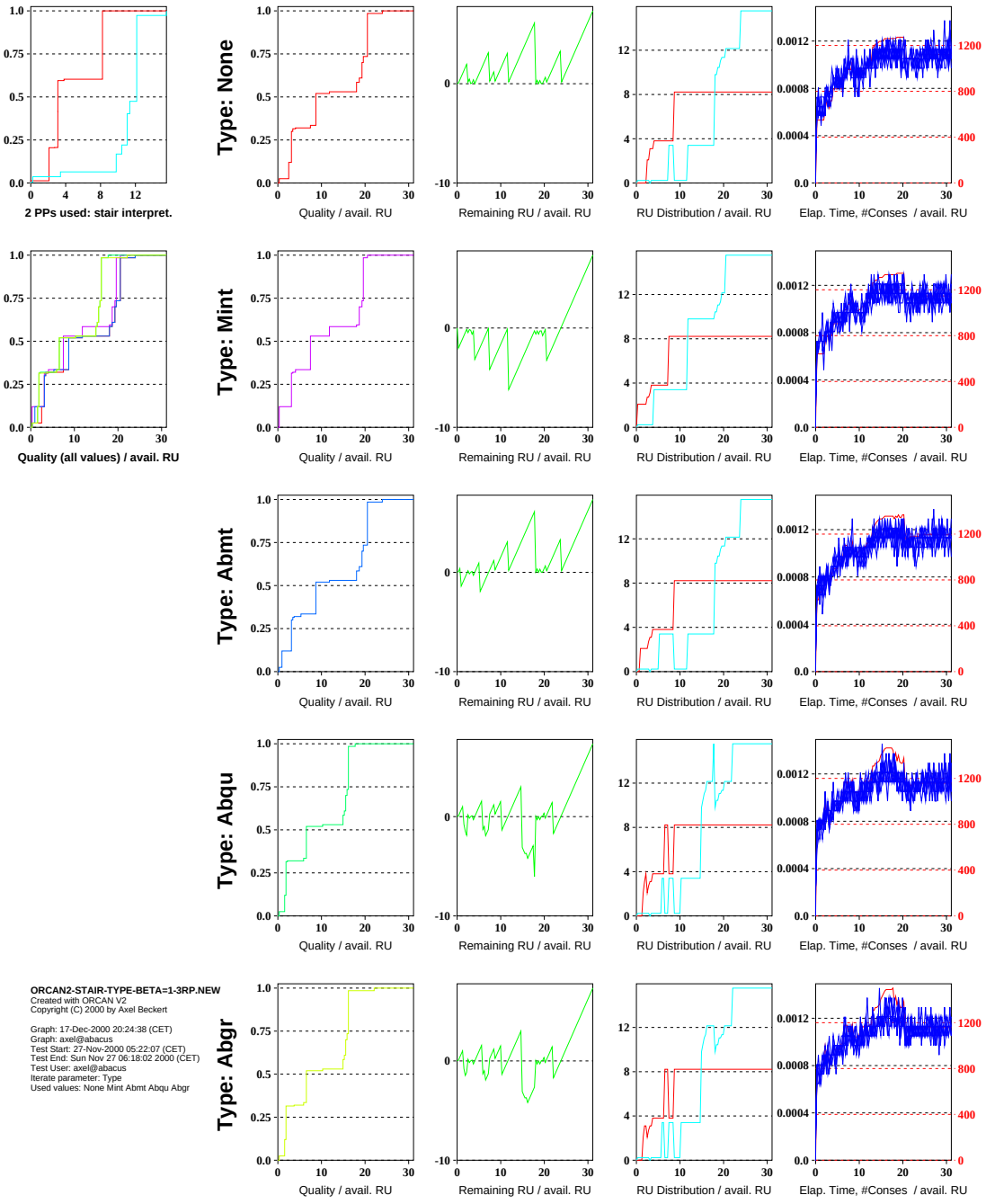

Abbildung A.26.: Laufzeittest mit 2 Performanzprofilen und Iteration des Treppenstufen-Methode-Parameters `:type` (bei $\alpha = 0$ und $\beta = 1$) über sämtliche möglichen Werte

A.2.2.3. Parameter `:alpha` und `:beta` der Treppenstufen-Methode

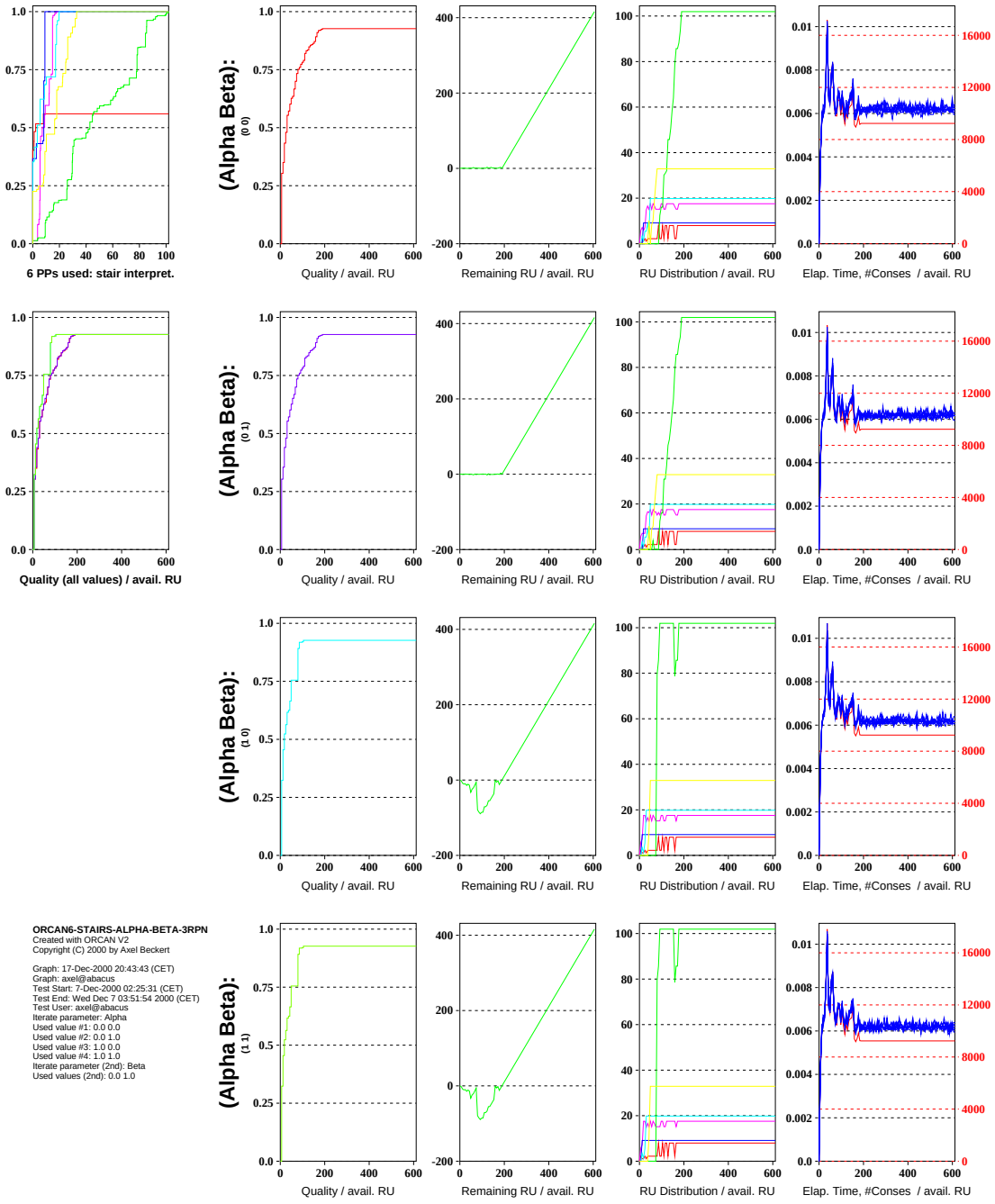

Abbildung A.27.: Laufzeittest mit 6 Performanzprofilen und Iteration der Parameter `:alpha` und `:beta` der Treppenstufen-Methode jeweils über die Werte 0 und 1

B. Interface

Für die Einbindung in andere Common-Lisp-Programme — also für die Kompilierung von Anytime-Algorithmen unter Zuhilfenahme ihrer Performanzprofile — sind die Funktionen `#'distribute` und `#'spp`[1] das zu verwendende Interface.

Für ORCAN-Benutzer stehen noch weiter die Funktionen `#'ps-page`[2] und `#'rtt`[3] zur Verfügung, um eigene Performanzprofile und Kompilierungsmethoden auf Laufzeit und Speicherverbrauch mit den verschiedenen Kompilierungsmethoden zu testen. Die Laufzeittests und -diagramme in dieser Arbeit wurden ebenfalls mit diesen beiden Funktionen erstellt.

Eine weitere in dieser Arbeit verwendete Funktion ist `#'random-profile`[4], welche zum Generieren der Beispiel-Performanzprofile verwendet wurde.

Der Quelltext der drei letztgenannten Funktionen ist in dieser Arbeit nicht abgedruckt, da diese Funktionen nur zur Laufzeitmessung von ORCAN selbst sowie zur Darstellung, aber nicht zur Berechnung von Ressourcenverteilungen nötig sind. Diese Funktionen finden sich in der Quelldatei `orcan-laufzeit.lisp` welche im WWW unter der Adresse http://w5.cs.uni-sb.de/~abe/ORCAN2/ erhältlich ist.

`distribute` *Funktion*

Syntax: `(distribute` *`time profiles [:key keyvalue ...]`*`)`

Aufgabe: Die Funktion `#'distribute` fungiert als Interface zu ORCAN und wird

[1] Interface ab Seite 142
[2] Interface ab Seite 145
[3] Interface ab Seite 143
[4] Interface ab Seite 147

verwendet zur Berechnung einer Ressourcenverteilung auf mehrere, durch Performanzprofile repräsentierte Anytime-Algorithmen.

Parameter:

***time*:** Anzahl der zur Verfügung stehenden Ressourceneinheiten

***profiles*:** Liste der zu verwendenden Performanzprofile

Optionale Keyword-Parameter

`:method` bestimmt die verwendete Berechnungsmethode:

- `:segment` Abschnittsweise lineare Funktionen *[Defaulteinstellung]*
- `:lin` Lineare Regression
- `:exp` Exponentielle Regression
- `:hill` Hillclimbing-Methode
- `:stair` Treppenstufen-Methode *[Defaulteinstellung]*

`:format` spezifiziert das Eingabeformat der übergebenen Performanzprofile:

- `:pp` Stützpunktlisten der Form (r-Wert, q-Wert) *[Defaulteinstellung]*
- `:abs` Relative Stützpunktlisten mit Stützpunkten der Form (relativer r-Wert, Steigung)
- `:lin` Parameter für lineare Funktionen: Steigung, q-Achsenabschnitt
- `:exp` Parameter für exponentielle Funktionen: η, λ
- `:func` Funktion oder Lambda-Ausdruck, welche eine monoton steigende Funktion $F : \mathbb{R}_0^+ \longmapsto [0;1]$ berechnet, den Ressourcenwert als Parameter übernimmt und die zu erwartende Qualität zurückgibt.
  Beispiele:
  - `#'f`
  - `#'(lambda (x) (f x))`

`:output` legt fest, ob das Ergebnis der Ressourcenverteilung als Absolutwerte oder als prozentualer Anteil an der insgesamt verteilten Ressourcenmenge zurückgegeben werden soll:

- `:absolute` Absolutwerte *[Defaulteinstellung]*
- `:percentage` prozentuale Anteile

`:combine` bestimmt die Funktion, mit der die zu erwartenden Qualitäten der einzelnen Performanzprofile kombiniert werden. Sie sollte die Abhängigkeiten der durch die Performanzprofile repräsentierten Anytime-Algorithmen darstellen. Neben

`#'+` Summierung aller Qualitäten *[Defaulteinstellung]*

`#'*` Multiplizierung aller Qualitäten[5]

kann auch jede andere Funktion verwendet werden, die $|\mathbb{P}|$ Parameter akzeptiert. Ein Beispiel hierfür wäre die Funktion `#'+*`[6].

Werden auf 1 normierte Ergebnisse gebraucht, sollte die per Keyword-Parameter `:combine` übergebene Funktion bereits normierte Ergebnisse zurückgeben.

Der Keyword-Parameter `:combine` wird nur von folgenden Kompilierungsmethoden verwendet, bei anderen Methoden hat er nur dann eine Wirkung, wenn von ORCAN das Verfahren bei sehr wenigen zu verteilenden Ressourcenmengen angewendet wird[7]:

- Treppenstufen-Verfahren
- Hillclimbing-Verfahren
- Verfahren bei wenig Ressourcen

`:accept-negative-time` legt fest, ob negative Werte des Parameters ***time*** akzeptiert und als Null betrachtet werden:

`t` Negative Werte des Parameters `time` werden akzeptiert und wie Null behandelt.

`nil` Negative Werte des Parameters `time` werden nicht akzeptiert und erzeugen eine Fehlermeldung. *[Defaulteinstellung]*

`:test-if-necessary` legt fest, ob Performanzprofile schon im voraus von der Berechnung ausgeschlossen werden, falls sie nur eine Qualität von Null erbringen, selbst wenn sie die komplette Ressourcenmenge zugewiesen bekommen.

`t` Performanzprofile, die selbst dann, wenn sie die komplette Ressourcenmenge zugewiesen bekommen, nur eine Qualität von Null erbringen, werden bereits im voraus von der Berechnung ausgeschlossen.
Warnung: Diese Option führt zu falschen Ergebnissen oder Abbrüchen, wenn über den Keyword-Parameter `:combine` eine Funktion übergeben wurde, welche nicht-triviale Abhängigkeiten zwischen den Anytime-Algorithmen darstellt oder nur genau $|\mathbb{P}|$ Parameter akzeptiert.

[5] Diese Einstellung ist nicht zu empfehlen, falls ein Performanzprofil für ein $r > 0$ eine Qualität von 0 zurückliefert, da dies bis dorthin in einer Gesamtqualität von 0% resultiert und ORCAN somit keine Qualitätssteigerung feststellen und damit auch keine sinnvolle Verteilung errechnen kann.

[6] Quellcode auf Seite 175 in der Quelldatei `orcan-helpers.lisp` (Abschnitt C.10)

[7] Siehe Abschnitt 5.4

`nil` Es werden alle Performanzprofile in die Kompilierung miteinbezogen. *[Defaulteinstellung]*

`:round` legt fest, ob und falls ja, wie stark die Ausgabe der errechneten Verteilung gerundet wird. Dies ist vorallem beim manuellen Auswerten oder Betrachten der Ergebnisse hilfreich.

`nil` Ergebnisse werden nicht gerundet. *[Defaulteinstellung]*

`<n>` Ergebnisse werden auf $n \in \mathbb{N}$ Nachkommastellen gerundet.

Optionale Keyword-Parameter für das Verfahren bei wenig Ressourcen:

Wie in Abschnitt 5.4 beschrieben, wählt ORCAN anhand der übergebenen Parameter ***time*** und ***profiles*** gegebenenfalls selbständig das Verfahren bei sehr wenigen zu verteilenden Ressourcenmengen. Entsprechend können die hier aufgeführten Keyword-Parameter bei allen Methoden angewendet werden; Wirkung zeigen sie aber nur, sofern ORCAN auch das Verfahren bei wenig Ressourcen als Kompilierungsmethode wählt.

`:distribute-on-small-interval` legt das Verhalten des Verfahrens bei wenig Ressourcen fest:

`:correct` Verwendet eine mathematisch korrekte Verteilung, d. h. verteilt die zu vergebende Menge an Ressourcen ***time*** gleichmäßig auf die Performanzprofile, die die größte anfängliche Steigung haben.

`:fair` Verwendet eine faire Verteilung, d. h. verteilt die zu vergebende Menge an Ressourcen ***time*** proportional zur anfänglichen Steigung der Performanzprofile.

`:display-warning` legt fest, ob eine Warnung ausgeben werden soll, falls die Keyword-Parameter-Kombination `:distribute-on-small-interval :correct` und `:combine #'*` verwendet wird.

`t` Die Warnung wird bei Ausführung der Funktion `#'small-interval`[8] ausgegeben.

`nil` Es wird keine Warnung diesbezüglich ausgegeben.

Optionale Keyword-Parameter für das Treppenstufen-Verfahren:

[8] Quellcode auf Seite 170 in der Quelldatei `orcan-small.lisp` (Abschnitt C.9)

Wird die Treppenstufen-Methode verwendet, also die Funktion `#'distribute` mit `(#'distribute` *`time profiles`* `:method :stair` *`[...]`*`)` aufgerufen, sind außerdem folgende Keyword-Parameter möglich[9].

`:repair-profiles` legt fest, ob die als Parameter übergebenen Performanzprofile vor der Verwendung auf fehlende Stützpunkte mit $x = 0$ überprüft und gegebenenfalls Punkte $(0 \mid 0)$ in die Profile eingesetzt sowie die für die Treppenstufen-Methode redundanten Punkte (siehe Abschnitt 4.3.2.1) entfernt werden.

- `t` Die übergebenen Performanzprofile werden gegebenenfalls geändert.
- `nil` Es werden keinerlei Änderungen an den übergebenen Performanzprofilen gemacht.

`:alpha` **und** `:beta` bestimmen das Intervall, in dem eine eventuelle Mehrvergabe an Ressourcen erlaubt ist. Wie das entsprechende Intervall aus diesen Werten berechnet wird, ist in den Formeln 5.1 und 5.2 in Abschnitt 5.5.2 wiedergegeben.

`:type` legt fest, nach welchem Ansatz die Mehrvergabe von Ressourcen geregelt wird (siehe auch Abschnitt 5.5.1).

- `:none` Es werden nicht mehr Ressourcen als ursprünglich gewollt vergeben.
- `:mint` Wenn der Algorithmus innerhalb der vorgegebenen Ressourcen keinen möglichen Schritt mehr findet, wird noch ein weiterer Schritt gemacht, und zwar bei demjenigen Anytime-Algorithmus, der für diesen zusätzlichen Schritt die kleinste Ressourcenmenge verbraucht.
- `:abmt` Wie `:mint`, jedoch mit einer durch die Keyword-Parameter `:alpha` und `:beta` bestimmten Grenze, die nicht überschritten werden darf.
- `:abgr` Innerhalb der durch die Keyword-Parameter `:alpha` und `:beta` bestimmten Grenze werden demjenigen Anytime-Algorithmus zusätzliche Ressourcen zugesprochen, welcher in einem zusätzlichen Schritt die effizienteste Qualitätssteigerung mit sich bringt.
- `:abqu` Innerhalb der durch die Keyword-Parameter `:alpha` und `:beta` bestimmten Grenze werden demjenigen Anytime-Algorithmus zusätzliche Ressourcen zugesprochen, welcher in einem zusätzlichen Schritt die größte Qualitätssteigerung mit sich bringt.

[9] Bei anderen Methoden haben sie keinerlei Wirkung.

Optionale Keyword-Parameter für das Gradientenverfahren:

Wird die Hillclimbing-Methode verwendet, also die Funktion `#'distribute` mit (`distribute` *`time profiles`* `:method :hill` *`[...]`*) aufgerufen, sind weiterhin folgende Keyword-Parameter möglich[9]:

`:step` ist die anfängliche Schrittweite. Werte kleiner als 10^{-5} sind aufgrund der Rechengenauigkeit von Liquid Common Lisp und diversen anderen Common-Lisp-Implementationen nicht zu empfehlen. Ebenfalls sollte der Wert einen bestimmten Bruchteil der zu vergebenden Ressourcenmenge nicht überschreiten. 10^{-1} ist die Defaulteinstellung.

`:tolerance-step` legt den Grenzwert für die Änderung der Schrittweite fest. Werte kleiner als 10^{-5} sind aufgrund der Rechengenauigkeit von Liquid Common Lisp und diversen anderen Common-Lisp-Implementationen nicht zu empfehlen. Ebenfalls sollte der Wert einen bestimmten Bruchteil der anfänglichen Schrittweite nicht überschreiten. 10^{-4} ist die Defaulteinstellung.

`:tolerance-diff` legt den Grenzwert für die Änderung der Qualität fest. Werte kleiner als 10^{-5} sind aufgrund der Rechengenauigkeit von Liquid Common Lisp und diversen anderen Common-Lisp-Implementationen nicht zu empfehlen. Ebenfalls sollte der Wert eindeutig kleiner als 1 sein, da der Algorithmus sonst im Zweifelsfall sofort und ohne irgendwelche sinnvollen Berechnungen terminiert. 10^{-4} ist die Defaulteinstellung.

`:tolerance` kann genutzt werden, um die beiden obigen Grenzwerte gleichzeitig auf einen gemeinsamen Grenzwert zu setzen. Sollen die beiden Grenzwerte getrennt angegeben werden, so kann dies durch die Angabe von `:tolerance :splitted` erreicht werden.

`:step-reduction` ist der Faktor, durch den die Schrittweite dividiert wird, falls sie verringert werden muß bzw. mit dem sie multipliziert wird, falls die Schrittweite vergrößert werden muß. Der Wert muß größer als 1 sein. Die Defaulteinstellung ist 2.

`:logical-operator` gibt die Verknüpfungsfunktion an, mit der die beiden Abbruchbedingungen verknüpft werden. Neben

`or` Oder-Verknüpfung *[Defaulteinstellung]*

`and` Und-Verknüpfung

kann theoretisch auch jede andere zweistellige boolesche Funktion verwendet werden. ORCAN V2 akzeptiert für diesen Keyword-Parameter sowohl das Symbol einer Funktion (z. B. `'f`) als auch eine Funktion

(z. B. `#'f`), wie dies auch die Common-Lisp-Funktion `#'apply` tut[10].

`:starting-point` gibt die initiale Ressourcenverteilung (den „Startpunkt“) an. Die Defaulteinstellung ist $\frac{time}{|\mathbb{P}|}$ Ressourceneinheiten für jedes Modul, also eine Gleichverteilung der zu vergebenden Ressourcenmenge auf alle Module. Diese Einstellung wird auch dann verwendet, wenn als Wert `:notgiven` übergeben wird.

`:always-test-dir` legt fest, ob der Hillclimbing-Algorithmus in jedem Schritt die zu wählende „Richtung“ neu berechnet, oder ob er die „Richtung“ so lange beibehält, bis sich in dieser „Richtung“ keine Verbesserung mehr findet.

`t` Die „Richtung“ wird in jedem Schritt neu bestimmt.

`nil` Die „Richtung“ wird erst neu bestimmt, wenn sich in der aktuellen „Richtung“ mit der aktuellen Schrittweite keine Verbesserung ergibt. *[Defaulteinstellung]*

Optionale Keyword-Parameter für das exponentielle Regressionsverf.:

Wird das exponentielle Regressionsverfahren verwendet, also die Funktion `#'distribute` mit `(distribute` *`time profiles`* `:method :exp` *`[...]`*`)` aufgerufen, ist weiterhin folgender Keyword-Parameter möglich[9]:

`:expx` muß eine streng monoton steigende Liste von r-Werten sein, welche die exponentielle Regressionsmethode als r-Werte verwendet, um die gemeinsamen Gesamtprofile zweier Performanzprofile darzustellen. Diese Daten sind allerdings nur notwendig, falls als Eingabeformat `:exp` gewählt wurde. (Defaulteinstellung: `'(0 1)`)

Rückgabewert: Das Resultat der Funktion sind drei Werte[11]. Dabei ist

der 1. Wert eine Liste, in der für jedes Anytime-Modul die Anzahl der Ressourceneinheiten, welche ihm zugeteilt wurden, aufgeführt sind. Die

[10] Da es sich in Common Lisp bei den beiden Funktionen `#'and` und `#'or` um Common-Lisp-Makros handelt, können diese nicht in der Form `(apply #'and` *`list`*`)` oder `(funcall #'and` *`argument`*$_1$ *`argument`*$_2$ `...)` verwendet werden [Steele, 1994, 7.3. Function Invocation]. Deswegen wird in ORCAN an den entsprechenden Stellen anstatt der Common-Lisp-Funktion `#'apply` eine Funktion `#'apply-ao` (für apply and or) verwendet. Sie macht nichts anderes, als die übergebene Funktion darauf zu testen, ob es eines der beiden Makros `#'and` oder `#'or` ist und falls ja, stattdessen die rekursiv definierten Funktionen `#'and-list` bzw. `#'or-list` aufzurufen, welche praktisch dieselbe Funktion wie die beiden Makros haben, aber nicht beliebig viele Parameter erwarten, sondern eine Liste mit den zu verknüpfenden Werten. (Entsprechend können sie auch nicht als einfaches `if`-Konstrukt verwendet werden.)

[11] in Form von Common-Lisp-Values

Reihenfolge entspricht der der übergebenen Performanzprofilliste.

der 2. Wert die approximierte Qualität des Ergebnisses, die bei dieser Verteilung zu erwarten ist.

der 3. Wert die Restressourcen. Dies sind die nicht verbrauchten Ressourcen, falls mehr Zeit verteilt werden sollte, als die Module benötigen, um ein optimales Ergebnis zu liefern, oder die zusätzlich vergebenen Ressourcen, falls eine Mehrvergabe berechnet wurde. In diesem Fall ist der Wert negativ. Bei Kompilierungsmethoden oder Eingabeformaten, bei denen keine Restressourcen berechnet können kann, ist dieser Wert 0.

Lisp-Package: `:orcan`

Quelldatei: `orcan-interface.lisp`[12]

---

spp *Funktion*

Syntax: `(spp` *`profiles [:key keyvalue ...]`*`)`

Aufgabe: Die Funktion `#'spp`[13] generiert die für Anytime-Module notwendigen System-Performanzprofile. Dazu iteriert sie über eine (durch Keyword-Parameter bestimmte oder anhand der übergebenen Performanzprofile errechnete) Menge von Ressourcenwerten sowie über einen — per Keyword-Parameter festgelegten — Keyword-Parameter der ORCAN-Funktion `#'distribute`.

Parameter:

***profiles*:** Liste der zu verwendenden Performanzprofile

Optionale Keyword-Parameter

Alle für die ORCAN-Funktion `#'distribute` gültigen Keyword-Parameter sind auch für die ORCAN-Funktion `#'spp` verfügbar. Zusätzlich kennt `#'spp` auch noch die folgenden Keyword-Parameter:

[12] Der Quelltext von `orcan-interface.lisp` ist im WWW unter der Adresse http://w5.cs.uni-sb.de/~abe/ORCAN2/ erhältlich.

[13] `#'spp` steht für „system performance profile“.

`:start`, `:in` **und** `:end` regeln die Iteration der Ressourcenwerte: `:start` legt den Startwert der Iteration fest (Defaulteinstellung: 0), `:in` die Schrittweite (Defaulteinstellung: 1) und `:end` die obere Grenze der Iteration (Defaulteinstellung: `nil`). Ist `:end` auf `nil` gesetzt, dann wird die Obergrenze für die Iteration dynamisch anhand der übergebenen Performanzprofile festgelegt.

Rückgabewert: Eine Liste aus Paaren mit Ressourcenwerten und den dazugehörigen Ergebnissen der Funktion `#'distribute`, bestehend aus zu erwartender Qualität, Ressourcenverteilung und Restressourcen.

Lisp-Package: `:orcan`

Quelldatei: `orcan-spp.lisp`[14]

`rtt` *Funktion*

Syntax: `(rtt` *`profiles [:key keyvalue ...]`*`)`

Aufgabe: Die Funktion `#'rtt`[15] ist gedacht, um Vergleiche von Laufzeit, Speicherverbrauch, Restressourcen und zu erwartender Qualität der verschiedenen ORCAN-Kompilierungsmethoden zu erstellen. Dazu iteriert `#'rtt` über per Keyword-Parameter festgelegte oder über die übergebenen Performanzprofile errechnete Menge von Ressourcenwerten und über einen — per Keyword-Parameter festgelegten — Keyword-Parameter der ORCAN-Funktion `#'distribute`. Die Ergebnisse von `#'rtt` können dann mit der ORCAN-Funktion `#'ps-page` grafisch dargestellt werden.

Parameter:

***profiles*:** Liste der zu verwendenden Performanzprofile

Optionale Keyword-Parameter

[14] Der Quelltext von `orcan-spp.lisp` ist im WWW unter der Adresse http://w5.cs.uni-sb.de/~abe/ORCAN2/ erhältlich.

[15] `#'rtt` steht für „run time test“.

Alle für die ORCAN-Funktion `#'spp` gültigen Keyword-Parameter sind auch für die ORCAN-Funktion `#'rtt` verfügbar. Zusätzlich kennt `#'rtt` auch noch die folgenden Keyword-Parameter:

`:rtt-file` Name der Datei, in der die errechneten Ergebnisse und die verwendeten Parameter abgelegt werden sollen. Die Defaulteinstellung ist `"example.rtt"`.

`:iterate` ist der `#'distribute`-Keyword-Parameter, über den iteriert werden soll. (Defaulteinstellung: `:method`)

`:values` ist die Liste der Werte, über die der durch den `#'rtt`-Keyword-Parameter `:iterate` festgelegte `#'distribute`-Keyword-Parameter iteriert werden soll. Die Defaulteinstellung ist `'(:stair :hill :lin :exp :segment)`

`:timerounds` gibt an, wie oft die Iteration über die zu vergebenden Ressourcenmengen für die Laufzeitmessung durchgeführt werden soll. Dabei werden die Werte nicht abschließend gemittelt, sondern die Werte aller Durchläufe gespeichert. Pro Durchlauf erscheint in der Ausgabe ein Graph im Laufzeitdiagramm. Je höher der Wert von `:timerounds` ist, desto höher ist die statistische Signifikanz der Laufzeitmessung für die jeweilige Performanzprofil-Kombination. (Defaulteinstellung: 8)

`:timeroundminval` Ist der Wert von `:timeroundminval` kleiner als $|\mathbb{P}|$, dann wird jeder einzelne Laufzeittest[16] $(timeroundminval - |\mathbb{P}|)$-mal durchgeführt und deren Laufzeit dann durch n dividiert. Die anderen Werte (errechnete Ressourcenverteilung, Restressourcen und Qualität sowie Speicherverbrauch) werden nach wie vor nur an einem Aufruf von `#'distribute` gemessen. (Defaulteinstellung: 16)

`:timeroundstairs` Im Fall von `:method :stair` wird jeder einzelne Laufzeittest $timeroundstairs \cdot (timeroundminval - |\mathbb{P}|)$-mal durchgeführt. (Defaulteinstellung: 8)

Rückgabewert: `nil`[17]

Lisp-Package: `:orcan`

Quelldatei: `orcan-laufzeit.lisp`[18]

[16] d. h. jeder Aufruf von `#'distribute`

[17] Die Ergebnisse finden sich in der über den Keyword-Parameter `:rtt-file` bestimmten Datei.

[18] Der Quelltext von `orcan-laufzeit.lisp` ist im WWW unter der Adresse http://w5.cs.uni-sb.de/~abe/ORCAN2/ erhältlich.

Beispiel: Um z. B. verschiedene Schrittweiten bei der Hillclimbing-Methode zu Vergleichen, würde sich der folgende Aufruf eigenen:

```
(#'rtt profiles
     :method :hill
     :iterate :step
     :values '(0.01 0.05 0.1 0.5))
```

ps-page *Funktion*

Syntax: (ps-page *rtt [:key keyvalue ...]*)

Aufgabe: Die Funktion `#'ps-page` liest die Daten eines Laufzeittests der ORCAN-Funktion `#'rtt` ein und generiert daraus eine Adobe-PostScript®-Datei mit Diagrammen.

Parameter:

***rtt*:** Dateiname der einzulesenden Datendatei oder die Daten in der Form, in der sie von `#'rtt` in eine Datei geschrieben wurden.

Optionale Keyword-Parameter

Über die optionalen Keyword-Parameter[19] läßt sich sowohl das Aussehen (z. B. Farben, Abstände, etc.) der Diagramme bestimmen, als auch, welche der möglichen Diagramme ausgegeben werden sollen.

`:ps-file` Name der Datei, unter die PostScript®-Diagramme abgelegt werden sollen. (Defaulteinstellung: `"example.ps"`)

`:order` Liste der zu zeichnenden Diagramme in der zu verwenden Reihenfolge. Erlaubte Werte sind:

[19] Es existieren noch weitere, hier bewußt nicht aufgeführte Keyword-Parameter. Diese haben die Funktion von Hilfsvariablen, da sie einerseits von anderen Keyword-Parametern abhängig sind und andererseits die Defaultwerte anderer Keyword-Parameter wiederum von ihnen abhängig sind. Aus diesem Grund reicht es nicht aus, sie mit dem Common Lisp Lambda-Listen-Schlüsselwort (Steele nennt die mit „`&`“ beginnenden Schlüsselwörter in der Parameterliste eines Common LispLambda-Ausdrucks oder einer Funktionsdefinition *„lambda-list keywords“* [Steele, 1994, 5.2.2. Lambda-Expressions].) `&aux` als Hilfsvariablen zu deklarieren.

`'quality`	Diagramm mit der zu erwartenden Qualität
`'remains`	Restressourcendiagramm
`'distribution`	Diagramm mit der errechneten Ressourcenverteilung für jedes der verwendeten Performanzprofile
`'profiles`	Diagramm mit den verwendeten Performanzprofilen
`'conses`	Speicherverbrauchsdiagramm
`'time`	Laufzeitdiagramm
`'conses-and-time`	Laufzeit- und Speicherverbrauchsdiagramm in einem Diagramm

Defaultwerte: Wurde der auszuwertende Laufzeittest unter Liquid Common Lisp gemacht, dann ist die Standardeinstellung

```
'(quality remains distribution profiles conses-
and-time)
```

Andernfalls ist die Standardeinstellung

```
'(quality remains distribution profiles)
```

`:mono` Für die Generierung von Farben, die eine unterschiedliche Helligkeit und Sättigung haben und damit auch auf Graustufen-Bildschirmen und -Druckern zu unterscheiden sind, muß der Wert `t` angegeben werden, für durchgehend gleichhelle, kräftige Farben `nil`. *[Defaulteinstellung]*

`:hsbmargins` Sofern der Keyword-Parameter `:mono` auf `t` gesetzt ist, beschreibt der Keyword-Parameter `:hsbmargins` den Mindestabstand von Sättigung und Helligkeit zu den jeweiligen Extremwerten Schwarz und Weiß bzw. 0 und 1. Sinnvoll sind Werte im Intervall $[0; 0{,}5[$.

`:colors` Liste von Farben[20], welche für die Darstellung der Performanzprofile und deren Ressourcenverteilung verwendet werden sollen. Die Liste sollte soviele Elemente haben wie Performanzprofile verwendet werden. Standardmäßig wird eine Liste von HSB[21]-Farben generiert, indem $|\mathbb{P}|$ verschiedene Farbtöne (*engl.* „Hue“) bei gleicher Sättigung (*engl.* „Saturation“) und Helligkeit (*engl.* „Brightness“) berechnet werden. Ist der Keyword-Parameter `:mono` auf `t` gesetzt, so werden auch verschiedene Helligkeit und Sättigungen verwendet.

`:qcolor` Farbe[20], in der das Diagramm mit der zu erwartenden Qualität gezeichnet werden soll. (Defaulteinstellung: `"blue"`)

[20] Als Werte sind Strings mit jeder Art von PostScript®-Farbformaten (z. B. die PostScript®-Funktionen `sethsbcolor` [Adobe Systems Inc., 1991b, Seite 506] und `setrgbcolor` [Adobe Systems Inc., 1991b, Seite 514] mit entsprechenden Parametern) sowie die vordefinierten Farben `"black"`, `"red"`, `"green"` und `"blue"` erlaubt.

[21] Hue, Saturation, Brightness

`:ccolor` Farbe[20], in der das Speicherverbrauchsdiagramm[22] gezeichnet werden soll. (Defaulteinstellung: `"red"`)

`:lcolor` Farbe[20], in der das Laufzeitdiagramm gezeichnet werden soll. (Defaulteinstellung: `"black"`)

`:rcolor` Farbe[20], in der das Restressourcendiagramm gezeichnet werden soll. (Defaulteinstellung: `"green"`)

`:strichelung` ist ein String, der die Werte eines PostScript®-Dash-Array (ohne die eckigen Klammern) für das PostScript®-Kommando `setdash` [Adobe Systems Inc., 1991b, Seite 500] enthalten sollte. (Defaulteinstellung: `"30 35"`)

`:numberofdashes` ist die gewünschte[23] Anzahl von Wertangaben an den Achsen der Diagramme. (Defaulteinstellung: 3)

`:maxc` **und** `:maxl` Obere Grenze[24] der `conses`- bzw. Laufzeitskala. (Defaulteinstellung: Wird anhand der darzustellenden Daten errechnet.)

Rückgabewert: `nil`[25]

Lisp-Package: `:orcan`

Quelldatei: `orcan-laufzeit.lisp`[26]

`random-profile` *Funktion*

Syntax: `(random-profile` *`[:key keyvalue ...]`*`)`

Aufgabe: Die Funktion `#'random-profile` generiert ein zufälliges Performanzprofil.

[22] Als Einheit für den Speicherverbrauch werden „conses“ verwendet, daher das „c“ im Namen des Keyword-Parameter.

[23] Um einigermaßen schöne Werte zu bekommen, kann es vorkommen, daß eine Angabe mehr oder eine weniger verwendet wird.

[24] Achtung: Graphen, die diese Grenze überschneiden, werden nicht abgeschnitten!

[25] Die erzeugten Diagramme finden sich in der über den Keyword-Parameter `:ps-file` bestimmten Datei.

[26] Der Quelltext von `orcan-laufzeit.lisp` ist im WWW unter der Adresse http://w5.cs.uni-sb.de/~abe/ORCAN2/ erhältlich.

Optionale Keyword-Parameter

Durch die optionalen Keyword-Parameter werden das Format und die minimale Länge sowie eventuelle Restriktionen bezüglich der Form (z. B. konvex, konkav, etc.) bestimmt.

`:format` legt das Format des zu generierenden Performanzprofils fest:

- `:pp` Zufällige, monoton steigende Stützpunktliste.
- `:lin` Zufällige Parameter für lineare Funktionen: Steigung, q-Achsenabschnitt
- `:lin-pp` Zufällige lineare Funktion in Form einer Stützpunktliste mit minimal nötiger Anzahl (zwei[27] oder drei) an Stützpunkten.
- `:exp` Parameter für exponentielle Funktionen: η, λ
- `:exp-pp` Zufällige exponentielle Funktion in Form einer Stützpunktliste mit einer zufälligen Anzahl[28] von Stützpunkten mit ebenfalls zufällig ausgewählten r-Werten.

`:minimum` Sofern das Rückgabeformat eine Stützpunktliste ist, so gibt dieser Wert das Minimum der Anzahl der Stützpunkte an. (Defaulteinstellung: 2)

`:maximum` Sofern das Rückgabeformat eine Stützpunktliste ist, so gibt dieser Wert das Maximum der Anzahl der Stützpunkte an. Der Wert `nil` bedeutet, daß es keine Beschränkung nach oben bei der Anzahl der Stützpunkte gibt. (Defaulteinstellung: nil)

Rückgabewert: Ein Performanzprofil entsprechend den durch die Keyword-Parameter festgelegten Spezifikationen.

Lisp-Package: `:orcan`

Quelldatei: `orcan-laufzeit.lisp`[29]

[27] Theoretisch ist bereits ein Punkt eine eindeutige Stützpunktliste, aber da für ORCAN an einigen Stellen mindestens zwei Stützpunkte notwendig sind, werden in Fällen, bei denen die Steigung 0 oder der q-Achsenabschnitt 1 ist, zwei Stützpunkte — $(0 \mid q)$ und $(1 \mid q)$ — zurückgegeben.

[28] Durch den Keyword-Parameter `:minimum` nach unten beschränkt.

[29] Der Quelltext von `orcan-laufzeit.lisp` ist im WWW unter der Adresse http://w5.cs.uni-sb.de/~abe/ORCAN2/ erhältlich.

C. Quelltext

Anmerkungen zum Quelltext

Der im folgenden abgedruckte Common Lisp Quelltext enthält die wichtigsten Funktionen der implementierten Kompilierungsmethoden. Er wurde mit LGrind [Various Artists, 1999] und PERL 5 [PERL, 2000] für die Einbindung in LaTeX-Dokumente formatiert. Dabei wurden zugunsten der Lesbarkeit u. a. folgende Umbenennungen vorgenommen:

- „`float-positive-infinity`“ wurde durch „∞“ ersetzt.
- In Wörtern ausgeschriebene Namen griechischer Buchstaben wurden durch die entsprechenden griechischen Buchstaben ersetzt, z. B. der Keyword-Parameter „`:alpha`“ durch „$:\alpha$“ und der Keyword-Parameter „`:beta`“ durch „$:\beta$“.
- Bei Variablen mit Indizes im Namen wurden diese tiefgestellt, z. B. wurde „`x1`“ durch „x_1“ ersetzt.
- Variablen, welche die Länge einer Liste oder eine Anzahl beinhalten, wurden in eine Schreibweise mit Betragsklammern umbenannt, z. B. von „`lp`“ (für „length-profiles“) in |`profiles`|.

Außerdem wird im Quelltext fast ausschließlich von der Ressource Zeit ausgegangen. Dies ist aber — wie bereits erwähnt — bezüglich der Verwendbarkeit von ORCAN für andere Arten von Ressourcen bedeutungslos.

Der komplette Common-Lisp-Quelltext von ORCAN V2 ist im World Wide Web unter der Adresse http://w5.cs.uni-sb.de/~abe/ORCAN2/ erhältlich.

C.1. Ausschnitte aus orcan-stairs.lisp

Die Quelldatei `orcan-stairs.lisp` beinhaltet die Funktionen für die Treppenstufen-Methode.

delete-redundant-points

```
(defun delete-redundant-points (profile)
  "Gibt eine Kopie des als Parameter uebergebenen Profils zurueck, bei welcher
die fuer die Treppenstufeninterpretation redundanten Stuetzpunkte (mit
gleichem Qualitaetswert) geloescht sind."
```

```
  (cond ((or (not (listp profile)))
         (error "orcan::delete-redundant-points: parameter must be a list of lists: ~A" profile))
        ((null profile) ())
        ((= (length profile) 1) profile)
        ((= (first (first profile)) (first (second profile)))
         (delete-redundant-points (cons (list (caar profile)
                                              (max (second (first profile))
                                                   (second (second profile))))
                                        (cddr profile))))
        ((= (second (first profile)) (second (second profile)))
         (delete-redundant-points (cons (car profile) (cddr profile))))
        (t (cons (car profile) (delete-redundant-points (cdr profile))))))
```

several-eql-maxima

```
(defun several-eql-maxima (maximum ml
                           &optional
                           ;;; Summe aller Zeitdifferenzen
                           (Σ_tmax 0)
                           ;;; Anzahl der bisher gefundenen Maxima.
                           (|max| 0))
  "Prueft, ob sich in der Liste der Maxima mehrere gleiche Maxima
befinden und, falls ja, ob sie zusammen in die noch zu vergebende
Zeit passen. Gibt die Anzahl der Maxima und die Summe der Zeiten der
Maxima zurueck."
  (let* ((maxtime (assoc maximum ml))
         (pos (position maxtime ml)))
    (cond ((equal pos nil)
           (list |max| Σ_tmax))
          (t (several-eql-maxima maximum
                                 (n^thcdr (1+ pos) ml)
                                 (+ Σ_tmax
                                    (- (car (third maxtime))
                                       (car (second maxtime))))
                                 (1+ |max|))))))
```

several-eql-minima

```
(defun several-eql-minima (minimum ml
                           &optional
                           ;;; Summe aller Zeitdifferenzen
                           (Σ_tmin 0)
                           ;;; Anzahl der bisher gefundenen Minima.
                           (|min| 0))
  "Prueft, ob sich in der Liste der Minima mehrere gleiche Minima
befinden und, falls ja, ob sie zusammen in die noch zu vergebende
Zeit passen. Gibt die Anzahl der Minima und die Summe der Zeiten der
Minima zurueck."
  (let* ((mintime (assoc minimum ml))
         (pos (position mintime ml)))
    (cond ((equal pos nil)
           (list |min| Σ_tmin))
          (t (several-eql-minima minimum
                                 (n^thcdr (1+ pos) ml)
                                 (+ Σ_tmin
                                    (- (car (third mintime))
                                       (car (second mintime))))
                                 (1+ |min|))))))
```

get-quality-diff

```
(defun get-quality-diff (time profile)
  "Gibt die Qualitaetssteigerung bei Zugabe von time Zeit an Profil
profile."
  (- (second (must-nround-assoc (+ time (caar profile)) profile))
     (second (first profile))))
```

find-best-of-several

```
(defun find-best-of-several (time maximum gl profiles
                             &key
                             (α 0)
                             (β 0)
                             (type :none)
                             (t_gesamt 0))
  "Wird von stairs-hill aufgerufen, falls es zwei oder mehr zum
aktuellen Zeitpunkt aequivalente Profile gibt."
  (loop for profile in profiles
        for pdata in gl
        for i from 0
        if (= (first pdata) maximum)
          collect (let* ((values (nth-stairs-hill i time gl profiles
                                                  :α α
                                                  :β β
                                                  :type type
                                                  :t_gesamt t_gesamt))
                         (distribution (loop for j from 0 to (1- (length profiles))
                                             collect (get-time-for-profile j values))))
                    (list (get-quality distribution profiles)
                          distribution))
            into all-possibilities
        finally (return (second (assoc (loop for q in all-possibilities
                                             maximize (first q))
                                       all-possibilities
                                       :test #'=)))))
```

find-best-of-several-lite

```
(defun find-best-of-several-lite (maximum gl &key (use-qual-instead-grad ()))
  "Wird von last-step aufgerufen, falls es zwei oder mehr zum
aktuellen Zeitpunkt aequivalente Profile gibt. Gibt Liste (profil-nr usedtime)
zurueck."
  (loop for p in gl
        for i from 0
        if (and (<= (first p) maximum) (plusp (first p)))
          collect (list (if use-qual-instead-grad
                            (diff_y (second p) (third p))
                          (gradient (second p) (third p)))
                        i
                        (diff_x (second p) (third p)))
            into all-possibilities
        finally (return (if (null all-possibilities)
                            ()
                          (let* ((pdata (assoc (loop for q in all-possibilities
                                                     maximize (first q))
                                               all-possibilities
                                               :test #'=)))
                            (list (second pdata) (third pdata)))))))
```

last-step

```
(defun last-step (time t_gesamt profiles
                  &key
                  (α 0)
                  (β 0)
                  (type :none))

  "Berechnet eine evtl. Mehrvergabe von Zeit. Gibt eine Liste mit
Elementen (profil-nr usedtime) zurueck. Dabei ist zusaetzlich
verwendbare Zeit (die erlaubte Ueberschreitung)

           overtime = time + (alpha * tges) + (beta * time)
                    = (alpha * tges) + ((1+beta) * time)
```

```
und die neue insgesamt zur Verfuegung stehende Zeit

            tgesneu = tges + (alpha * tges) + (beta * time)
130                 = ((1+alpha) * tges) + (beta * time).

Dabei ist

overtime = ab demnaechst noch zu verbratende Zeit,
tges     = bisher insgesamt zu vergebende Zeit (entspricht time auf
           Toplevel-Ebene),
tgesneu  = ab demnaechst insgesamt zu vergebende Zeit,
time     = bisher noch zu vergebende Zeit (entspricht time auf lokaler
           Ebene innerhalb der Iterationen),
140 tver     = bereits verbrauchte Zeit,
         = tges - time,

Beispiele für Kombinationen von alpha und beta:
              (0 0) => Entspricht :none
              (1 0) => Ergebnis in Intervall [0 2*Tges]
              (0 1) => Ergebnis in Intervall [0 Tges+time], d.h. es darf
                       noch doppelt soviel Zeit verbraucht werden, wie
                       uebrig ist.
"
150   (cond ((eq type :none) ()) ;;; Keine Ueberschreitung der Zeitgrenze
        ((eq type :mint)     ;;; Minimalste Ueberschreitung
         (let* ((gl (loop for p in profiles
                          and i from 0
                          collect (list (diff_x (first p)
                                                (second p))
                                        (first p)
                                        (second p))))
                (erg (find-best-of-several-lite (apply #'minnotzero
                                                       (mapcar #'first
160                                                            gl))
                                                gl)))
           (if (null erg)
               ()
             (list erg))))
        ((eq type :abmt)     ;;; Minimalste Ueberschreitung innerhalb
                             ;;; der von alpha und beta gesetzten
                             ;;; Grenzen.
         (let* ((overtime (+ time (* α t_gesamt) (* β time)))
                ;;; overtime = noch zu verbratende Zeit.
170             (glx (loop for p in profiles
                           and i from 0
                           if (<= (diff_x (first p) (second p))
                                  (+ time overtime))
                             collect (list (diff_x (first p)
                                                   (second p))
                                           (first p)
                                           (second p)) into gl1
                           collect (list (diff_x (first p)
                                                 (second p))
180                                      (first p)
                                         (second p)) into gl2
                           finally return (list gl1 gl2)))
                (gl (second glx))
                (gls (first glx))
                (mingl (apply #'minnil (mapcar #'first gls)))
                (erg (if (null mingl) ()
                         (find-best-of-several-lite mingl gl))))
           (if (null erg)
               ()
```

```
190             (list erg))))
          ((member type '(:abgr :abqu))
           ;;; Ueberschreitung in durch alpha und beta festgelegtem
           ;;; Intervall, bei abgr wird der Punkt mit der groessten
           ;;; Steigung genommen, bei abqu, der mit der hoechsten
           ;;; Qualitaetssteigerung.
           (let* ((overtime (+ time (* α t_gesamt) (* β time)))
                  ;;; overtime = noch zu verbratende Zeit.
                  (gl (loop for p in profiles
                            collect (search-best-gradient overtime p
                                                          :use-qual-instead-grad
200                                                       (eq type :abqu))))
                  (erg (find-best-of-several-lite (apply #'max
                                                         (mapcar #'first
                                                                 gl))
                                                  gl)))
             (if (null erg)
                 ()
               (list erg))))
          (t (error "No valid type: ~A" type))))
210

(defun search-best-gradient (time profile &key (use-qual-instead-grad ()))                search-best-gradient
  "Sucht die Stuetzpunktkombination mit der groessten Steigung
dazwischen, die noch innerhalb des Zeitlimits liegt. Gibt eine
Liste aus mit: Steigung, erstem Punkt im Profil und erstem Punkt
im Profil, welcher die hoechste Steigung gegenueber dem ersten
Punkt aufweist. Keine Ueberpruefung der Parameter!"
  (cond ((null (cdr profile))
         (list -1 '(-1 -1) '(-1 -1)))
        (t (let* ((gl (mapcar
220                     #'(lambda (x)
                            (cond ((<= (- (car x) (caar profile)) time)
                                   (list (if use-qual-instead-grad
                                             (diff_y (car profile) x)
                                           (gradient (car profile) x))
                                         x))
                                  (t (list -1 '(-1 -1)))))
                         (cdr profile)))
                  (maximum (apply #'max (mapcar #'car gl))))
             (list maximum (car profile) (second (assoc maximum gl)))))))
230

(defun nth-stairs-hill (n time gl profiles                                                nth-stairs-hill
                        &key
                        (α 0)
                        (β 0)
                        (type :none)
                        (t_gesamt 0))
  "Errechnet aus den von stairs-hill uebergebenen Daten das endgueltige
Ergebnis fuer diesen Schritt und ruft stairs-hill mit dem Rest wieder auf.
Verwendet das n-te (mit Null zu zaehlen beginnend) Profil."
240   (let* ((oldprofiledata (n^th n gl))
         (oldprofile (n^th n profiles))
         (new-x (first (third oldprofiledata)))
         (point-pos (position (assoc new-x oldprofile) oldprofile))
         (newprofile (n^thcdr point-pos oldprofile))
         (usedtime (- new-x (first (second oldprofiledata))))
         (newtime (- time usedtime))
         (newprofiles (return-n^th-changed profiles n newprofile)))
    (append (list (list n usedtime))
            (stairs-hill newtime newprofiles
250                         :changed-profile n
                            :gl gl
```

```
                        :α α
                        :β β
                        :type type
                        :t_gesamt t_gesamt))))

(defun stairs-hill (time profiles                                    stairs-hill
                    &key
                    (changed-profile -1)
260                 (gl ())
                    (α 0)
                    (β 0)
                    (type :none)
                    (t_gesamt 0))
  "Gibt eine Liste von Paaren (Profil-Nr. Verwendete-Zeit) zurueck.
Keine Ueberpruefung der Parameter! Bitte vorher ein (profile-repair
profiles) machen!"
  (cond ((= changed-profile -1)
         ;;; gl muss initialisiert werden, da eine loop-Schleife ueber
270      ;;; die Laenge der kuerzesten Liste geht.
         (setq gl (make-list (length profiles)))))
  (cond ((= time 0) ()) ;;; last-step wird hier nicht gebraucht, da das
                        ;;; Ergebnis bereits optimal ist.
        (t (setq gl (loop for y in profiles
                          and x in gl
                          and z from 0
                          collect (cond ((= changed-profile z)
                                         (search-best-gradient time y))
                                        ((= changed-profile -1)
280                                      (search-best-gradient time y))
                                        ((= (car x) -1)
                                         x)
                                        ((< (diff_x (second x) (third x)) time)
                                         x)
                                        ;;; Sonst:
                                        ;;; Berechne nur das verwendete neu.
                                        (t (search-best-gradient time y)))))
                ;;; gl = Liste der Punkte mit groesster Steigung
                ;;;      innerhalb der Zeit. Kein Punkt gefunden ->
290             ;;;      Steigung = -1
           (let* ((maximum (apply #'max (mapcar #'car gl)))
                  ;;; maximum = maximale in gl vorkommende steigung
                  (seqmax (several-eql-maxima maximum gl)))
             (cond ((minusp maximum)
                    (last-step time t_gesamt profiles
                               :α α
                               :β β
                               :type type))
                   ((and (> (second seqmax) time) (>= (first seqmax) 2))
300 ;;; Gibt es zwei Profile, die in Frage kommen, aber ich kann nur noch
    ;;; die Zeit auf eines der beiden verteilen, dann muss ich in einer
    ;;; Extra-Funktion entscheiden, welches ich verwende.
                    (find-best-of-several time maximum gl profiles
                                          :α α
                                          :β β
                                          :type type
                                          :t_gesamt t_gesamt))
    ;;; Ansonsten kann ich wie gewohnt weiter machen.
                   (t (nth-stairs-hill (position (assoc maximum gl) gl)
310                                    time
                                       gl
                                       profiles
                                       :α α
                                       :β β
```

```
                                    :type type
                                    :t_gesamt t_gesamt)))))))
```

```
(defun get-time-for-profile (profile-no distribution &optional (sum 0))                get-time-for-profile
  "Summiert die fuer Profil #profile-no vergebene Zeit auf und gibt
320 eine Liste mit Profil-Nummer und ihm zuzuweisender Zeit zurueck."
  (let ((comb (assoc profile-no distribution)))
    (cond ((null comb) (list profile-no sum))
          (t (get-time-for-profile profile-no
                                   (cdr (member comb distribution :test #'equal))
                                   (+ sum (second comb)))))))
```

```
(defun sh-distribution (time profiles                                          sh-distribution
                            &key
                            (combine #'+)
330                         (α 0)
                            (β 0)
                            (type :none)
                            (repair-profiles t))
  "Gibt Liste zurueck: Beste Zeitverteilung, zu erwartende
Mindestqualitaet und uebrige Zeit."
  (let* ((repaired-profiles (if repair-profiles
                                (profile-repair profiles)
                                profiles))
         (init-distribution (stairs-hill time repaired-profiles
340                                      :α α
                                         :β β
                                         :type type
                                         :t_gesamt time))
         (distribution (loop for i from 0 to (1- (length profiles)) collect
                             (get-time-for-profile i init-distribution)))
         (usedtimeoverall (loop for j in distribution sum (second j)))
         (quality (apply combine
                         (mapcar #'(lambda (x y)
                                     (second (must-nround-assoc (second x)
350                                                             y)))
                                 distribution
                                 repaired-profiles))))
    (values (mapcar #'second distribution)
            (nround quality)
            (- time usedtimeoverall))))
```

C.2. Ausschnitte aus orcan-hill.lisp

Die Quelldatei `orcan-hill.lisp` beinhaltet die Funktionen für die Hillclimbing-Methode.

```
(defun new-coord (n xliste direction step &aux verteilung)                     new-coord
  "Errechnet den Vektor, wenn man von Vektor xliste aus mit Schritt
step in Richtung direction geht."
  (cond ((/= (length xliste) n)
         (error "(new-coord) xliste=~A ist nicht ~D-dimensional"
                xliste n))
```

```
        ((or (< direction 1) (> direction n))
         (error "(new-coord) direction=~A ist nicht zwischen 1 und ~D"
                direction n))
        (t (let ((summe (- (apply '+ xliste)
                           (nth (1- direction) xliste))))
             ; summe aller elemente ohne direction.
          (if (= 0 summe)
              ;;; Falls nix mehr zum Verteilen da ist, gebe die
              ;;; alte Liste zurueck
              xliste
            ;;; Sonst berechne neu.
            (progn
              (setq step (min step summe)
                    verteilung (mapcar #'(lambda (x) (/ x summe)) xliste))
                 ; prozentualer Wert der Elemente von xliste im Verhaeltnis
                 ; zu summe
              (setf (nth (1- direction) verteilung) -1)
                 ; Das Element 'direction' wird auf -1 gesetzt,
                 ; damit step hinzuaddiert anstatt, dass ein
                 ; Bruchteil davon abgezogen wird.
              (mapcar #'(lambda (x prozentsatz)
                          (- x (* prozentsatz step)))
                      xliste
                      verteilung)))))))


(defun quality-for-direction (n direction q xliste step)                    quality-for-direction
  "Gibt die zu erwartende Qualitaet zurueck, bei einem Schritt in die
Richtung direction."
  (cond ((not (numberp direction))
         (error "(hill-n:quality-for-direction) ~A ist keine Nummer"
                direction))
        ((or (< direction 1) (> direction n))
         (error "(hill-n:quality-for-direction) ~A ist nicht zwischen 1 und ~D"
                direction n))
        (t (funcall q (new-coord n xliste direction step)))))


(defun enlarge-step (max_t xliste step step-reduction)                      enlarge-step
  "Setzt die Schrittweite hoeher."
  (min (- max_t (apply 'max xliste)) (* step step-reduction)))


(defun find-direction (max_t n q xliste step step-reduction                 find-direction
                       &aux maxwert)
  "Sucht die guenstigste Richtung zum Weitersuchen."
  (setq maxwert (list (quality-for-direction n 1 q xliste step) 1))
  (loop for i from 2 to n do
        (let ((actual_quality (quality-for-direction n i q xliste step)))
          (if (> actual_quality (car maxwert))
              (setq maxwert (list actual_quality i)))))
  (if (zerop (car maxwert))
      (let ((new-step (enlarge-step max_t xliste step step-reduction)))
        (cond ((= new-step step) (list 0 'exit))
              (t (find-direction max_t n q xliste
                                 (enlarge-step max_t xliste step step-reduction)
                                 step-reduction))))
    maxwert))


(defun linear-f (p2g-liste)                                                 linear-f
  "Wandelt die Ausgabe von punkte2gerade oder linear-regression in
eine normalisierte Funktion um.  p2g-liste: eine Liste von Steigung
und y-Achsenabschnitt Gibt ein Funktion der folgenden Form zurueck:
```

```
      _____
   ___/  ^---- y=1
70  ^---- y=0
"
  (let* ((m (car p2g-liste))
         (a (cadr p2g-liste))
         (max_x (if (zerop m) 'infinite (/ (- 1 a) m))))
    #'(lambda (x)
      (cond ((minusp x) 0)
            ((eq max_x 'infinite) a)
            ((<= x max_x) (let ((y (+ a (* m x))))
                            (if (minusp y) 0 y)))
80          (t 1)))))


(defun lin-ausschnitt (footing-list x)                                  lin-ausschnitt
  "Sucht aus einem Performanzprofil und einem x Wert rekursiv die
passende Steigung mit y-Achsenabschnitt.  Ausgabe im punkte2gerade
Format (list m y0)."
  (let ((lfl (length footing-list)))
    (cond
     ;;; Fehler falls Footing-Liste zu klein
     ((= lfl 0)
90
      (error "Ich ziehe mich in die Ecke zurueck und schmolle: ~
              (lin-ausschnitt) Dieses Perfoschwanz-Profil hat nur ~
              keinen Stuetzpunkt: ~A" footing-list))
     ;;; Fehler falls Footing-Liste nur ein Element und dessen x-Wert
     ;;; groesser 0, sonst '(0 y-Wert) zurueckgeben.
     ((= (length footing-list) 1)
      (list 0 (cadar footing-list)))
     ;;; Falls x vor dem naechsten Stuetzpunkt liegt, gebe Gerade
     ;;; zwischen erstem und zweitem Stuetzpunkt aus.
100  ((< x (caadr footing-list))
      (let ((sfl (subseq footing-list 0 2)))
        (punkte2gerade (car sfl) (cadr sfl))))
     ;;; Falls x hinter dem naechsten Stuetzpunkt liegt, suche weiter
     ;;; im Rest der Footing-Liste
     ((> x (caadr footing-list))
      (lin-ausschnitt (cdr footing-list) x))
     ;;; Fuer alle folgenden Faelle ist (= x (caadr footing-list))
     ;;; wahr:

110  ;;; Ist die Laenge der Footing-Liste 2, so ist der zweite Punkt
     ;;; der letzte. Entsprechend nehmen wir dahinter eine waagrechte
     ;;; Gerade an.
     ((= (length footing-list) 2)
      (let ((p2g (punkte2gerade (car footing-list)
                                (cadr footing-list))))
        (list (/ (car p2g) 2)
              (/ (cadr p2g) 2))))
     ;;; Ist der zweite Stuetzpunkt nicht der letzte, dann gebe den
     ;;; Schnitt der vorherigen und der darauffolgenden Gerade aus.
120  (t (let* ((sfl_1 (subseq footing-list 0 2))
               (sfl_2 (subseq footing-list 1 3))
               (p2g_1 (punkte2gerade (car sfl_1) (cadr sfl_1)))
               (p2g_2 (punkte2gerade (car sfl_2) (cadr sfl_2))))
          (list (/ (+ (car p2g_1)   (car p2g_2))   2)
                (/ (+ (cadr p2g_1) (cadr p2g_2)) 2)))))))


(defun pp-lin-f (footing-list)                                          pp-lin-f
  "Wandelt eine Footing-Liste (Liste von Stuetzpunkt-Paaren) in eine
```

```
    Funktion um. Diese ist linear zwischen den einzeln Stuetzpunkten."
130   #'(lambda (x)
          (if (minusp x)
              0
            (funcall (linear-f (lin-ausschnitt footing-list x)) x))))


    (defun combine-quality-n-f (fliste &key (combine #'+))                    combine-quality-n-f
      "Gibt eine Funktion Q(x) zurueck, wobei Q(x) die Qualitaet der
    Funktionen aus der Liste 'fliste' angewendet auf den Vektor x und
    danach verknuepft sind.  :combine : Verknuepfungsfunktion, im
    Normalfall #'+ oder #'*."
140   (cond ((not (listp fliste))
             (error "(combine-quality-n-f) Parameter ~A ist keine Liste" fliste))
            (t (let* ((n (length fliste)))
                     #'(lambda (xliste)
                         (cond ((not (listp fliste))
                                (error "(combine-quality-n-f)Parameter ~A ~
                                          ist keine Liste" xliste))
                               ((> n (length xliste))
                                (error "(combine-quality-n-f) Liste ~A hat ~
                                          weniger als ~D Elemente" xliste n))
150                            (t (apply combine
                                         (mapcar #'(lambda (f c)
                                                     (max 0 (min 1 (funcall f c))))
                                                 fliste
                                                 xliste)))))))))


    (defun hill-n-rek (max_t n q xliste direction step actual_quality          hill-n-rek
                           logical-operator step-reduction
                           tolerance-diff tolerance-step always-test-dir)
      "Rekursiver Teil von hill-n: Das eigentliche 'Klettern'... :->"
160   (cond
       ((null direction)
        (hill-n-rek max_t n q xliste (cadr (find-direction max_t n q xliste step
                                                           step-reduction))
                    step (funcall q xliste) logical-operator
                    step-reduction tolerance-diff tolerance-step
                    always-test-dir))
       ((equal direction 'exit)
        (append xliste (list actual_quality)))
       ((not (numberp direction))
170     (error "(hill-n-rek) ~A ist keine Nummer" direction))
       ((> tolerance-step (abs (- (apply '+ xliste)
                                  (n^th (1- direction) xliste))))
        (append xliste (list actual_quality)))
       (t (let* ((expected_quality (if always-test-dir
                                       0
                                     (quality-for-direction n direction q xliste
                                                            step)))
                 ;;; Die zu erwartende Qualitaet wuerde
                 ;;; die Richtung beibehalten.
180              (new_direction_and_quality
                  (cond ((and (> expected_quality actual_quality)
                              (not always-test-dir))
                         (list direction expected_quality))
                        (t (let* ((test (find-direction max_t n q xliste step
                                                        step-reduction))
                                  (test_quality (car test))
                                  (test_direction (cadr test)))
                             (cond ((> test_quality actual_quality)
                                    (list test_direction test_quality))
190                                 (t (setq step (/ step step-reduction))
```

```
                                (list  direction
                                       (quality-for-direction n direction q
                                                              xliste
                                                              step))))))))))
                        ;;; Ist die zu erwartende Qualitaet niedriger als die
                        ;;; aktuelle, so wird die von diesem Punkt aus beste
                        ;;; Richtung gesucht. Ist keine Richtung besser als der
                        ;;; aktuelle Punkt, so wird die Schrittweite reduziert und
                        ;;; in die urspruengliche Richtung weiter gegangen.
                        ;;; Nach dieser Berechnung wird, wenn noetig, die zu
                        ;;; erwartende Qualitaet neu berechnet.
                (cond ((apply-ao logical-operator ;;; Logischen Operator anwenden.
                                 ;;; Grenzwerte vergleichen
                                 (> tolerance-step step)
                                 (> tolerance-diff
                                    (abs (- actual_quality
                                            (cadr new_direction_and_quality)))))
                       (append xliste (list actual_quality)))
                              ;;; Sobald beide Grenzwerte unterschritten
                      (t (hill-n-rek max_t
                                     n
                                     q
                                     (new-coord n
                                                xliste
                                                (car new_direction_and_quality)
                                                step)
                                     (car new_direction_and_quality)
                                     step
                                     (cadr new_direction_and_quality)
                                     logical-operator
                                     step-reduction
                                     tolerance-diff
                                     tolerance-step
                                     always-test-dir)))))))
```

```
(defun hill-n (max_t n profiles                                                hill-n
               &key
               (combine #'+)
               (starting-point :notgiven)
               (step 0.1)
               (tolerance-diff 0.001)
               (tolerance-step 0.001)
               (tolerance :splitted)
               (step-reduction 2)
               (logical-operator 'or)
               (format :pp)
               (no-values nil)
               (always-test-dir nil)
               &aux
               fliste)
  "hill-n findet *ein* lokales Qualitaetsmaximum fuer n
Performanzprofile Rueckgabe: eine Liste von Zeitwerten/der
Zeitverteilung, deren Summe wieder maxT ergeben muss."
  (cond ((member format '(:pp :sl))
         (setq fliste (mapcar #'pp-lin-f profiles)))
        ((member format '(:lin :linear))
         (setq fliste (mapcar #'(lambda (x)
                                  (lin2func (first x) (second x)))
                              profiles)))
        ((member format '(:exp :exponential))
         (setq fliste (mapcar #'(lambda (x)
                                  (exp2func (first x) (second x)))
                              profiles)))
```

```
        ((member format '(:func :lambda)
         (setq fliste profiles)))
        (t (error "Unknown format (~A) to orcan:hill-n." format)))
  (if (not (eq :splitted tolerance)) (setq tolerance-diff tolerance
                                           tolerance-step tolerance))
  (let ((erg (cond ((not (listp fliste))
                    (error "(hill-n) Parameter ~A ist keine Liste" fliste))
                   ((/= (length fliste) n)
                    (error "(hill-n) Parameter-Liste ~A hat nicht die Laenge ~D"
                           fliste n))
                   ((zerop max_t)
                    (make-list (1+ n) :initial-element 0))
                   (t (if (eq starting-point :notgiven)
                          (setq starting-point
                                (make-list n :initial-element (/ max_t n))))
                      (hill-n-rek max_t
                                  n
                                  (combine-quality-n-f fliste :combine combine)
                                  starting-point
                                  ()
                                  step
                                  0
                                  logical-operator
                                  step-reduction
                                  tolerance-diff
                                  tolerance-step
                                  always-test-dir)))))
;;; Hier gab's urspruenglich noch die Normierung bei Verwendung der
;;; Verknuepfungsfunktion +. Diese ist in die Funktion distribute
;;; abgewandert, wo sie auch die Normierung anderer Methoden
;;; vollzieht. Deswegen eine hundsgewoehnliche Rueckgabe:
    (if no-values
        erg
      (values (butlast erg)
              (car (last erg))))))
```

C.3. Ausschnitte aus orcan-reglin.lisp

Die Quelldatei `orcan-reglin.lisp` beinhaltet die Funktionen für die lineare Regression.

linear-regression

```
(defun linear-regression (profil)
  "Die Funktion linear-regression fuehrt eine lineare Regression
durch. Rueckgabewert ist das Paar (Steigung y-Achsenabschnitt)."
  (let ((Σ_xiyi (Σlist (mapcar #'car profil)))
        (Σ_yi (Σlist (mapcar #'cadr profil)))
        (Σ_xiyi (Σlist (mapcar #'(lambda (x) (* (car x) (cadr x))) profil)))
        (Σ_xi^2 (Σlist (mapcar #'(lambda (x) (sqr (car x))) profil)))
        (n (length profil)))
    (list (/ (- (* n Σ_xiyi) (* Σ_xiyi Σ_yi))
             (- (* n Σ_xi^2) (sqr Σ_xiyi)))
          (/ (- (* Σ_xi^2 Σ_yi) (* Σ_xiyi Σ_xiyi))
             (- (* n Σ_xi^2) (sqr Σ_xiyi))))))
```

t1-lin

```
(defun t1lin (gesamtzeit a1 q1 a2 q2)
  "Berechnet t1(T) nach ORCAN V1 Doku Seite 5, Formel 9
incl. Exception-Handling."
  (let ((y (cond ((= 0 a1) 0)
                 ((= 0 a2) gesamtzeit)
                 (t (/ (+ (- gesamtzeit
                             (/ q1 a1))
                          (/ q2 a2))
                       2)))))
;;; die Zeit fuer das 1. Modul darf nicht <0 oder >Gesamtzeit sein.
;;;
;;; Fuer die Gerade a*x+q ist (1-q)/a der x-Wert, bei dem die Gerade
;;; den Wert 1 hat; fuer unsere Profile heisst das, dass es sich
;;; nicht lohnt, einem der Profile mehr als diese Zeit zuzuteilen!
    (let ((c (/ (- 1 q1) a1))
          (diffx (- gesamtzeit (/ (- 1 q2) a2))))
      (cond ((> y c) (setq y c))
            ((< y diffx) (setq y diffx))))
    (cond ((> y gesamtzeit) (setq y gesamtzeit))
          ((minusp y) (setq y 0)))
    y))
```

zeitverteilung-2-lin

```
(defun zeitverteilung-2-lin (zeit profil1 profil2 &key
                                 (format :pp))
  "Zeitverteilung-2-lin berechnet *nur* die Zeitverteilung fuer 2
Profile sowie die zu erwartende Gesamtqualitaet. Das kombinierte
Performanzprofil liefert die Funktion gesamtprofil-2-lin."
  (let* ((gerade1 (cond ((member format '(:sl :pp))
                         (linear-regression profil1))
                                 ;;; Profile in Geraden
                                 ;;; umwandeln.
                        ((member format '(:lin :linear)) profil1)
                                 ;;; Profile haben bereits
                                 ;;; richtiges Format
                        (t (error "zeitverteilung-2-lin only accepts the formats \":pp\" and \":lin\",
you tried to use the \"%A\" format."
                                  format))))
         (gerade2 (cond ((member format '(:sl :pp))
                         (linear-regression profil2))
                        ((member format '(:lin :linear)) profil2)))
         (a1 (first gerade1))
         (q1 (second gerade1))
         (a2 (first gerade2))
         (q2 (second gerade2))
         ;;; Berechne Zeitzuteilung fuer erstes Modul
         (zeit1 (t1lin zeit a1 q1 a2 q2)))
;;; Rueckgabe der beiden Zeiten sowie der geschaetzten Qualitaet als
;;; values.
    (values (list zeit1 (- zeit zeit1))
            (* (gerade gerade1 zeit1) (gerade gerade2 (- zeit zeit1))))))
```

C.4. Ausschnitte aus orcan-treelin.lisp

Die Quelldatei `orcan-treelin.lisp` beinhaltet die Funktionen für die Erstellung von Binärbäumen der linearen Regressionsmethode.

```
(defun gesamtprofil-n-lin-2 (profile &key (format :pp))                    gesamtprofil-n-lin-2
  "gesamtprofil-n-lin-2 berechnet den Gesamtprofil fuer n Profile mit
der linear-regressiven Methode.  Dabei wird die Liste der Profile in 2
Haelften geteilt, fuer die rekursiv je ein Gesamtprofil berechnet
wird; und die resultierenden Profile werden mit Hilfe der Funktion
fuer 2 Profile kombiniert.

Rueckgabewerte haben immer die Form von Stuetzpunktlisten.
"
  (cond ((null profile) NIL)
        ((= 1 (length profile)) (cond ((member format '(:lin :linear))
                                       (linear2stuetz (first profile)))
                                      ((member format '(:pp :sl))
                                       (first profile)))) ; 1 P. - zurueckgeben
        ((= 2 (length profile)) ; 2 P. - direkt kombinierbar
         (gesamtprofil-2-lin (first profile) (second profile) :format format))
        (t
         (let ((firsthalf (butlast profile (floor (/ (length profile) 2))))
                                                                  ; p1,...,p[n/2]
               (lasthalf (nthcdr (ceiling (/ (length profile) 2)) profile)))
                                                                  ; p[n/2+1],...,pn
           (gesamtprofil-2-lin (gesamtprofil-n-lin-2 firsthalf :format format)
                               (gesamtprofil-n-lin-2 lasthalf :format format)
                               :format :pp)))))


(defun zeitverteilung-n-lin-2 (zeit profile &key (format :pp))             zeitverteilung-n-lin-2
  "zeitverteilung-n-lin-2 berechnet die Zeitverteilung fuer n Profile
mit der linear-regressiven Methode und geht dazu folgendermassen vor:
  - Spalte die Liste der Eingabeprofile in 2 Haelften.
  - Berechne je ein Gesamtprofil fuer die beiden Haelften.
  - Verteile die Zeit auf die beiden Profile aus Schritt 2, um die
    Gesamtzeiten fuer jede der Haelften zu ermitteln.
  - Verteile die Zeiten aus Schritt 3 auf ihre entsprechenden Haelften.
"
  (cond ((null profile) (values NIL 0))
        ((= 1 (length profile)) ; 1 P. bekommt volle Zeit
         (values (list zeit)
                 (gerade (cond ((member format '(:sl :pp))
                                (linear-regression (first profile)))
                               ((member format '(:lin :linear))
                                (first profile)))
                         zeit)))
        ((= 2 (length profile)) ; 2 P. - direkt kombinierbar
         (zeitverteilung-2-lin zeit (first profile) (second profile)
                               :format format))
        (t
         (let* ((firsthalf (butlast profile (floor (/ (length profile) 2))))
                                                                  ; p1,...,p[n/2]
                (lasthalf (nthcdr (ceiling (/ (length profile) 2)) profile))
                                                                  ; p[n/2+1],...,pn
                (profil1 (gesamtprofil-n-lin-2 firsthalf :format format))
                                                                  ; Das Gesamtprofil von p1,...,p[n/2]
                (profil2 (gesamtprofil-n-lin-2 lasthalf :format format))
                                                                  ; Das Gesamtprofil von p[n/2+1],...,pn
```

```
                (zeit-1-2 (zeitverteilung-2-lin zeit profil1 profil2
                                                :format :pp))
                                          ; Die Zeiten fuer jede Haelfte
                (zv1 NIL)
                (exQ1 0)
                (zv2 NIL)
                (exQ2 0))
          (multiple-value-setq (zv1 exQ1)
                               (zeitverteilung-n-lin-2 (first zeit-1-2)
                                                       firsthalf
                                                       :format format))
                               ; Berechne endgueltige Verteilung fuer die erste...
          (multiple-value-setq (zv2 exQ2)
                               (zeitverteilung-n-lin-2 (second zeit-1-2)
                                                       lasthalf
                                                       :format format))
                               ; ... und die zweite Haelfte
          (values (append zv1 zv2)
                  (* exQ1 exQ2))))))
```

```
(defun gesamtprofil-2-lin (profil1 profil2 &key (format :pp))                gesamtprofil-2-lin
  "Die Funktion gesamtprofil-2-lin liefert zu 2 Profilen das
geschaetzte Gesamtprofil in Form von Stuetzpunktlisten."
  (let* ((gerade1 (cond ((member format '(:sl :pp))
                         (linear-regression profil1))
                        ((member format '(:lin :linear)) profil1)))
         (gerade2 (cond ((member format '(:sl :pp))
                         (linear-regression profil2))
                        ((member format '(:lin :linear)) profil2)))
         (a1 (first gerade1))
         (q1 (second gerade1))
         (a2 (first gerade2))
         (q2 (second gerade2)))
    ;;; Berechnung des Performanzprofiles der zusammengesetzten Funktion:
    (mapcar #'(lambda (x)
                (let* ((t1 (t1^lin x a1 q1 a2 q2))
                       (y (* (gerade gerade1 t1) (gerade gerade2 (- x t1)))))
                  (list x y)))
            (merge-lists (stuetzpunkte profil1 :format format)
                         (stuetzpunkte profil2 :format format)))))
```

C.5. Ausschnitte aus orcan-regexp.lisp

Die Quelldatei `orcan-regexp.lisp` beinhaltet die Funktionen für die exponentielle Regression.

```
(defun exp-regression (profil)                                               exp-regression
  "exp-regression fuehrt eine exponentielle Regression durch mit dem
Ansatz
                        y = 1 - (eta * e^(-lambda * x))

Rueckgabewert ist die Liste (lambda eta)."
  (let* ((backup (copy-tree profil)) ;;; Profil sichern...
         (params (linear-regression
```

```
                (mapcar #'(lambda (x) (setf (cadr x)
                                            (log (- 1
                                                    (* 0.999 (cadr x))))) x)
                                          ;;; um 1.0 zu umgehen: Profil stauchen
                        backup))))
              ;;; forme das profil so um, dass eine Linearregression
              ;;; durchgefuehrt werden kann
    (list (- (first params)) (my-exp (second params)))))


(defun t_1^exp (gesamtzeit l_1 n_1 l_2 n_2)                                    t1-exp
  "Berechnet t1(T) nach ORCAN V1 Doku Seite 7, Formel 15
incl. Exception-Handling."
  (let ((y (/ (+ (- (log (* l_1 n_1))
                    (log (* l_2 n_2)))
                 (* gesamtzeit l_2))
              (+ l_2 l_1))))
;;; die Zeit fuer das 1. Modul darf nicht <0 oder >Gesamtzeit sein.
    (cond ((> y gesamtzeit) (setq y gesamtzeit))
          ((minusp y) (setq y 0)))
    y))


(defun zeitverteilung-2-exp (zeit profil_1 profil_2 &key (format :pp))         zeitverteilung-2-exp
 "Zeitverteilung-2-exp berechnet *nur* die Zeitverteilung fuer 2
Profile sowie die zu erwartende Gesamtqualitaet. Das kombinierte
Performanzprofil liefert die Funktion gesamtprofil-2-exp."
 (let* ((expo_1 (cond ((member format '(:sl :pp))
                       (exp-regression profil_1))
                                       ;;; Profile in e-Funktionen
                                       ;;; umwandeln.
                      ((member format '(:exp :exponential))
                       profil_1)
                                         ;;; Profile haben bereits
                                         ;;; richtiges Format
                      (t (error "zeitverteilung-2-exp only accepts the formats \":pp\" and \":exp\",
you tried to use the \"%A\" format."
                                format))))
        (expo_2 (cond ((member format '(:sl :pp))
                       (exp-regression profil_2))
                      ((member format '(:exp :exponential))
                       profil_2)))
        (l_1 (first expo_1))
        (n_1 (second expo_1))
        (l_2 (first expo_2))
        (n_2 (second expo_2))
;;; Berechne Zeitzuteilung fuer erstes Modul
        (zeit_1 (t_1^exp zeit l_1 n_1 l_2 n_2)))
;;; gib beide Zeiten sowie die geschaetzte Qualitaet zurueck
   (values (list zeit_1 (- zeit zeit_1))
           (/ (+ (expfunkt zeit_1 expo_1)
                 (expfunkt (- zeit zeit_1) expo_2))
              2))))
```

C.6. Ausschnitte aus orcan-treeexp.lisp

Die Quelldatei `orcan-treeexp.lisp` beinhaltet die Funktionen für die Erstellung von Binärbäumen der exponentiellen Regressionsmethode.

gesamtprofil-n-exp-2

```
(defun gesamtprofil-n-exp-2 (profile &key
                                     (format :pp)
                                     (exp_x '(0 1)))
  "gesamtprofil-n-exp-2 berechnet den Gesamtprofil fuer n Profile mit
der exponentiell-regressiven Methode.  Dabei wird die Liste der
Profile in 2 Haelften geteilt, fuer die rekursiv je ein Gesamtprofil
berechnet wird; und die resultierenden Profile werden mit Hilfe der
Funktion fuer 2 Profile kombiniert."
  (cond ((null profile) NIL)
        ((= 1 (length profile)) (cond ((member format '(:exp :exponential))
                                       (exp2stuetz (first profile)
                                                   :exp_x exp_x))
                                      ((member format '(:pp :sl))
                                       (first profile)))) ; 1 P. - zurueckgeben
        ((= 2 (length profile)) ; 2 P. - direkt kombinierbar
         (gesamtprofil-2-exp (first profile) (second profile)
                             :format format
                             :exp_x exp_x))
        (t
         (let ((firsthalf (butlast profile (floor (/ (length profile) 2))))
                                                                ; p1,...,p[n/2]
               (lasthalf (n^thcdr (ceiling (/ (length profile) 2)) profile)))
                                                                ; p[n/2+1],...,pn
           (gesamtprofil-2-exp (gesamtprofil-n-exp-2 firsthalf
                                                     :format format
                                                     :exp_x exp_x)
                               (gesamtprofil-n-exp-2 lasthalf
                                                     :format format
                                                     :exp_x exp_x)
                               :format :pp
                               :exp_x exp_x)))))
```

zeitverteilung-n-exp-2

```
(defun zeitverteilung-n-exp-2 (zeit profile &key
                                    (format :pp)
                                    (exp_x '(0 1)))
"zeitverteilung-n-exp-2 berechnet die Zeitverteilung fuer n Profile
mit der exponentiell-regressiven Methode und geht dazu
folgendermassen vor:
 - Spalte die Liste der Eingabeprofile in 2 Haelften.
 - Berechne je ein Gesamtprofil fuer die beiden Haelften.
 - Verteile die Zeit auf die beiden Profile aus Schritt 2, um die
   Gesamtzeiten fuer jede der Haelften zu ermitteln
 - Verteile die Zeiten aus Schritt 3 auf ihre entsprechenden Haelften.
"
  (cond ((null profile) (values NIL 0))
        ((= 1 (length profile)) ; ein Profil - kriegt ganze Zeit
         (values (list zeit)
                 (expfunkt zeit
                           (cond ((member format '(:sl :pp))
                                  (exp-regression (first profile)))
                                 ((member format '(:exp :exponential))
                                  (first profile))))))
        ((= 2 (length profile)) ; 2 P. - direkt kombinierbar
         (zeitverteilung-2-exp zeit (first profile) (second profile)
```

```
                              :format format))
          (t
           (let* ((firsthalf (butlast profile (floor (/ (length profile) 2))))
                                        ; p1,...,p[n/2]
                  (lasthalf (nthcdr (ceiling (/ (length profile) 2)) profile))
                                        ; p[n/2+1],...,pn
                  (profil1 (gesamtprofil-n-exp-2 firsthalf
                                                 :format format
                                                 :expx expx))
                                        ; Das Gesamtprofil von p1,...,p[n/2]
                  (profil2 (gesamtprofil-n-exp-2 lasthalf
                                                 :format format
                                                 :expx expx))
                                        ; Das Gesamtprofil von p[n/2+1],...,pn
                  (zeit-1-2 (zeitverteilung-2-exp zeit profil1 profil2
                                                  :format :pp))
                                        ; Die Zeiten fuer jede Haelfte
                  (zv1 NIL)
                  (exQ1 0)
                  (zv2 NIL)
                  (exQ2 0))
             (multiple-value-setq (zv1 exQ1)
                                  (zeitverteilung-n-exp-2 (first zeit-1-2) firsthalf
                                                          :format format
                                                          :expx expx))
                                   ; Berechne endgueltige Verteilung fuer die erste...
             (multiple-value-setq (zv2 exQ2)
                                  (zeitverteilung-n-exp-2 (second zeit-1-2) lasthalf
                                                          :format format
                                                          :expx expx))
                                   ; ... und die zweite Haelfte
             (values (append zv1 zv2)
                     (arith-mittel exQ1 exQ2))))))


(defun gesamtprofil-2-exp (profil1 profil2 &key                          gesamtprofil-2-exp
                                   (format :pp)
                                   (expx '(0 1)))
  "Die Funktion gesamtprofil-2-exp liefert zu 2 Profilen das
geschaetzte Gesamtprofil in Form von Stuetzpunktlisten."
  (let* ((expo1 (cond ((member format '(:sl :pp)) (exp-regression profil1))
                      ((member format '(:exp :exponential)) profil1)))
         (expo2 (cond ((member format '(:sl :pp)) (exp-regression profil2))
                      ((member format '(:exp :exponential)) profil2)))
         (l1 (first expo1))
         (e1 (second expo1))
         (l2 (first expo2))
         (e2 (second expo2)))
    ;;; Berechnung des Performanzprofiles der zusammengesetzten Funktion:
    (mapcar #'(lambda (x)
                (let* ((t1 (t1^exp x l1 e1 l2 e2))
                       (y (/ (- 2
                                (/ e1 (my-exp (* l1 t1)))
                                (/ e2 (my-exp (* l2 (- x t1)))))
                             2)))
                       ;;; Jetzt koennte es allerdings sein, dass der
                       ;;; Wert ueber 1 oder unter 0 liegt. Das muessen
                       ;;; wir abfangen!
                  (cond ((> y 1) (setq y 1))
                        ((minusp y) (setq y 0)))
                  (list x y)))
            (if (member format '(:exp :exponential))
                expx
              (merge-lists (stuetzpunkte profil1 :format format)
```

```
                    (stuetzpunkte profil2 :format format))))))
```

C.7. Ausschnitte aus orcan-segment.lisp

Die Quelldatei `orcan-segment.lisp` beinhaltet die Funktionen für die abschnittsweise lineare Methode.

```
(defun spiegel (profil zeit &optional (result NIL))                                    spiegel
"'spiegel' spiegelt 'profil' an der y-Achse und verschiebt es um
'zeit' nach rechts."
  (cond ((null profil) result)
        (t (spiegel (rest profil) zeit (cons (list (- zeit (caar profil))
                                                   (cadar profil))
                                             result)))))
```

```
(defun lin-interpol (punkt1 punkt2 x)                                                  lin-interpol
"lin-interpol interpoliert linear zwischen 2 Punkten:

  y1-y2
 ------- * (x-x1) + y1
  x1-x2
"
  (let ((x1 (car punkt1))
        (y1 (cadr punkt1))
        (x2 (car punkt2))
        (y2 (cadr punkt2)))
    (+ (* (/ (- y1 y2)
             (- x1 x2))
          (- x x1))
       y1)))
```

```
(defun abschnitt-fkt (profil x &optional (prev (first profil)))                        abschnitt-fkt
  "abschnitt-fkt berechnet den Wert, den die durch 'profil' definierte
Funktion aus abschnittsweisen Geraden an der Stelle x hat."
  (cond ((null profil) (cadr prev))  ;;; falls Profil zuende, liefere letztes y
        ((<= x (caar profil))
         (cond ((> x (car prev)) (lin-interpol prev (car profil) x))
               ;;; x liegt zwischen letztem und naechstem x-Wert des Profils
               (t (cadar profil))))  ;;; falls x kleiner als kleinster
                        ;;; x-Wert des Profils, gib zugehoerigen y-Wert zurueck
        (t (abschnitt-fkt (rest profil) x (first profil)))))
```

```
(defun kombi (profil1 profil2 kombfkt)                                                 kombi
  "kombi verknuepft 2 Profile mittels eines Operators, der als
Argument beim Aufruf mit uebergeben wird."
  (mapcar #'(lambda (x) (list x (funcall kombfkt
                                         (abschnitt-fkt profil1 x)
                                         (abschnitt-fkt profil2 x))))
          (merge-lists (mapcar #'car profil1)
                       (mapcar #'car profil2))))
```

findmax

```
(defun findmax (profil &optional
                       (max_x^aktuell (caar profil))
                       (max_y^aktuell (cadar profil)))
"findmax findet das (x y) Paar mit dem maximalen Y-Wert."
  (let ((aktuelles (first profil)))
    (if (null profil)
        (list max_x^aktuell max_y^aktuell)
      (if (> (second aktuelles) max_y^aktuell)
          (findmax (rest profil) (first aktuelles) (second aktuelles))
        (findmax (rest profil) max_x^aktuell max_y^aktuell)))))
```

zeitverteilung-2-absch

```
(defun zeitverteilung-2-absch (zeit profil_1 profil_2
                                    &optional (kombfkt #'arith-mittel))
  "Die Hauptfunktion der abschnittsweise linearen Methode. Sie nimmt 2
Profile, die zu verteilende Zeit und die Funktion, mit der die Profile
kombiniert werden sollen; dafuer liefert sie eine liste mit der Zeit
fuer das 1. Modul, der Zeit fuer das 2. Modul und die zu erwartende
Gesamtqualitaet."
  (let ((t_1 (first (findmax (kombi profil_1 (spiegel profil_2 zeit) kombfkt)))))
    (cond ((minusp t_1) (setq t_1 0))
          ((> t_1 zeit) (setq t_1 zeit)))
    (values (list t_1 (- zeit t_1))
            (funcall kombfkt (abschnitt-fkt profil_1 t_1)
                     (abschnitt-fkt profil_2 (- zeit t_1))))))
```

C.8. Ausschnitte aus orcan-treeabs.lisp

Die Quelldatei `orcan-treeabs.lisp` beinhaltet die Funktionen für die Erstellung von Binärbäumen der abschnittsweise linearen Methode.

gesamtprofil-n-absch-2

```
(defun gesamtprofil-n-absch-2 (profile &optional (kombfkt #'arith-mittel))
  "gesamtprofil-n-absch-2 berechnet das Gesamtprofil fuer n Profile
mit der abschnittsweise linearen Methode.  Dabei wird die Liste der
Profile in 2 Haelften geteilt, fuer die rekursiv je ein Gesamtprofil
berechnet wird; und die resultierenden Profile werden mit Hilfe der
Funktion fuer 2 Profile kombiniert.

Rueckgabewerte haben immer in Form von Stuetzpunktlisten."
  (cond ((null profile) NIL)
        ((= 1 (length profile)) (first profile))   ; 1 P. - zurueckgeben
        ((= 2 (length profile))                    ; 2 P. - direkt kombinierbar
         (gesamtprofil-2-absch (first profile) (second profile) kombfkt))
        (t
         (let ((firsthalf (butlast profile (floor (/ (length profile) 2))))
                                                   ; p1,...,p[n/2]
               (lasthalf (n^thcdr (ceiling (/ (length profile) 2)) profile)))
                                                   ; p[n/2+1],...,pn
           (gesamtprofil-2-absch (gesamtprofil-n-absch-2 firsthalf kombfkt)
                                 (gesamtprofil-n-absch-2 lasthalf kombfkt)
                                 kombfkt)))))
```

zeitverteilung-n-absch-2

```
(defun zeitverteilung-n-absch-2 (zeit profile &optional
                                      (kombfkt #'arith-mittel))
```

```
  "zeitverteilung-n-lin-2 berechnet die Zeitverteilung fuer n Profile
mit der abschnittsweise linearen Methode und geht dazu folgendermassen
vor:
  - Spalte die Liste der Eingabeprofile in 2 Haelften.
  - Berechne je ein Gesamtprofil fuer die beiden Haelften.
  - Verteile die Zeit auf die beiden Profile aus Schritt 2, um die
    Gesamtzeiten fuer jede der Haelften zu ermitteln.
  - Verteile die Zeiten aus Schritt 3 auf ihre entsprechenden Haelften.
"
  (cond ((null profile) (values NIL 0))
        ((= 1 (length profile))                    ; 1 P. bekommt volle Zeit
         (values (list zeit)
                 (abschnitt-fkt (first profile) zeit)))
                                                   ; ein Profil - kriegt ganze Zeit
        ((= 2 (length profile))                    ; 2 P. - direkt kombinierbar
         (zeitverteilung-2-absch zeit (first profile) (second profile)
                                 kombfkt))
        (t
         (let* ((firsthalf (butlast profile (floor (/ (length profile) 2))))
                                                   ; p1,...,p[n/2]
                (lasthalf (nthcdr (ceiling (/ (length profile) 2)) profile))
                                                   ; p[n/2+1],...,pn
                (profil1 (gesamtprofil-n-absch-2 firsthalf kombfkt))
                                                   ; Das Gesamtprofil von p1,...,p[n/2]
                (profil2 (gesamtprofil-n-absch-2 lasthalf kombfkt))
                                                   ; Das Gesamtprofil von p[n/2+1],...,pn
                (zeit-1-2 (zeitverteilung-2-absch zeit profil1 profil2
                                                  kombfkt))
                                                   ; Die Zeiten fuer jede Haelfte
                (zv1 NIL)
                (exQ1 0)
                (zv2 NIL)
                (exQ2 0))
           (multiple-value-setq (zv1 exQ1)
                                (zeitverteilung-n-absch-2 (first zeit-1-2)
                                                          firsthalf kombfkt))
                                 ; Berechne endgueltige Verteilung fuer die erste...
           (multiple-value-setq (zv2 exQ2)
                                (zeitverteilung-n-absch-2 (second zeit-1-2)
                                                          lasthalf kombfkt))
                                 ; ... und die zweite Haelfte
           (values (append zv1 zv2)
                   (funcall kombfkt exQ1 exQ2))))))


(defun gesamtprofil-2-absch (profil1 profil2                          gesamtprofil-2-absch
                             &optional
                             (kombfkt #'arith-mittel))
  "Die Funktion gesamtprofil-2-absch liefert zu 2 Profilen das
geschaetzte Gesamtprofil in Form von Stuetzpunktlisten."
  ;;; Berechnung des Performanzprofiles der zusammengesetzten Funktion:
  (mapcar #'(lambda (x)
              (let ((y 0)
                    (dummy nil))      ;;; jetzt berechnen wir den
                                      ;;; Funktionswert zu diesem x-Wert
                (multiple-value-setq (dummy y)
                                     (zeitverteilung-2-absch x profil1 profil2
                                                             kombfkt))
                (list x y)))
          (merge-lists (stuetzpunkte profil1 :format :pp)
                       (stuetzpunkte profil2 :format :pp))))
```

C.9. Ausschnitte aus orcan-small.lisp

Die Quelldatei `orcan-small.lisp` beinhaltet die Funktionen für die Methode bei sehr kleinen Ressourcenmengen.

first-gradient

```
(defun first-gradient (profile)
  "Gibt die Steigung zwischen den ersten beiden Stuetzpunkten zurueck."
  (cond ((not (listp profile)) (error "Profile is no list: ~A" profile))
        ((< (length profile) 2)
         (error "Profile needs at least 2 bases: ~A" profile))
        ((eq (caadr profile) (caar profile))
         (error "Base 1 and 2 have the same x-value: ~a and ~a"
                (car profile) (cadr profile)))
        (t (/ (- (cadadr profile) ; y2
                 (cadar profile)) ; y1
              (- (caadr profile)  ; x2
                 (caar profile))) ; x1
        )))
```

verteile-sif

```
(defun verteile-sif (max_t verteilung)
  "Verteilt maxT im Verhaeltnis der Liste verteilung und gibt eine Liste
gleicher Laenge zurueck. (Faire Verteilung)"
  (let ((faktor (/ max_t (apply '+ verteilung)))) ; sum*faktor=maxT
    (mapcar #'(lambda (x) (* x faktor)) verteilung)))
```

verteile-sic

```
(defun verteile-sic (max_t verteilung)
  "Verteilt maxT im Verhaeltnis der Liste verteilung und gibt eine Liste
gleicher Laenge zurueck. (Korrekte Verteilung)"
  (let* ((maximum (apply 'max verteilung))
         (v_2 (substitute-if 0
                             #'(lambda (x) (not (eq x maximum)))
                             verteilung))
         (faktor (/ max_t (apply '+ v_2)))) ; sum*faktor=maxT
    (mapcar #'(lambda (x) (* x faktor)) v_2)))
```

quality-si

```
(defun quality-si (gradienten-liste verteilung
                                  &key
                                  (combine '+))
  "Berechnet die zu erwartende Qualitaet bei zu vergebender Zeit maxT
und kleinen Intervallen."
    (apply combine (mapcar #'* gradienten-liste verteilung)))
```

small-interval

```
(defun small-interval (max_t profiles
                           &key
                           (distribute 'fair)
                           (display-warning t)
                           (combine '+))
  "Berechnet eine Zeitverteilung anhand der Steigung zwischen den
ersten beiden Stuetzpunkten. Man kann zwischen mathematisch korrekter
Verteilung (:distribute 'correct) und fairer Verteilung (:distribute
'fair) waehlen. Desweiteren kann man mit dem Keyword :combine
beeinflussen, welche Verknuepfungsfunktion zur Berechnung der zu
erwartenden Qualtitaet verwendet werden soll."
  (if (and (member combine (list #'* '*))
           (member distribute (list 'correct :correct))
           display-warning)
```

```
      (format t "
**********************************************************************
This is a WARNING from orcan:small-interval. You use \":combine '*\"
and \":distribute 'correct\" in this function together. This may cause
a resulting quality of zero (which will probably confuse the functions
using this value), if not all profiles have the same first gradient!
**********************************************************************

You may disable this warning by passing the value nil to the key
\":display-warning\" in this function.
"))
  (let* ((gradienten-liste (mapcar #'first-gradient profiles))
         (verteilung (cond ((member distribute (list 'fair :fair))
                            (verteile-sif max_t gradienten-liste))
                           ((member distribute (list 'correct :correct))
                            (verteile-sic max_t gradienten-liste))
                           (t (error
                               "The distribution type need to be either 'fair or 'correct, but is: ~A"
                               distribute)))))
    (values verteilung (quality-si gradienten-liste verteilung
                                   :combine combine))))
```

C.10. Ausschnitte aus orcan-helpers.lisp

Die Quelldatei `orcan-helpers.lisp` beinhaltet verschiedene Hilfsfunktionen für die verschiedenen Methoden.

diffx
```
(defun diff_x (p_1 p2g)
  "Errechnet die Differenz der x-Werte zweier Punkte."
  (if (or (null (first p_1)) (null (first p2g)))
      0
    (abs (- (first p2g) (first p_1)))))
```

diffy
```
(defun diff_y (p_1 p2g)
  "Errechnet die Differenz der y-Werte zweier Punkte."
  (if (or (null (second p_1)) (null (second p2g)))
      0
    (abs (- (second p2g) (second p_1)))))
```

gerade
```
(defun gerade (parameter x)
  "Die Funktion gerade nimmt als Parameter die Parameter einer
Gerade (Steigung und y-Achsenabschnitt) und einen x-Wert und rechnet
trivialerweise den y-Wert dazu aus... (aber beschraenkt ihn auf Werte
im Intervall [0,1])."
  (let ((result (+ (* x (first parameter)) (second parameter))))
    (cond ((> result 1) (setq result 1))
          ((minusp result) (setq result 0)))
    result))
```

expfunkt
```
(defun expfunkt (x params)
  "Die Funktion gerade nimmt als Parameter die Parameter einer
expnonetiellen Funktion (lambda und eta) und einen x-Wert und rechnet
```

```
trivialerweise den y-Wert dazu aus... (aber beschraenkt ihn auf Werte
im Intervall [0,1])."
  (let ((y (- 1 (* (second params) (my-exp (* -1 (first params) x))))))
    (cond ((minusp y) (setq y 0))
          ((> y 1) (setq y 1)))
    y))


(defun punkte2gerade (eins zwei)                                    punkte2gerade
"Die Funktion punkte2gerade wandelt zwei Punkte in eine Gerade
(bzw. deren notwendige Parameter, die Steigung und den
y-Achsenabschnitt) um.

    Eingabeparameter: ((x1 y1) (x2 y2)).

Dabei sollte x2 > x1 sein, sonst wird die Steigung negiert.

    Ausgabe: (Steigung y-Achsenabschnitt).

Wenn x1=x2, dann wird x2 ein winziges Stueckchen heraufgesetzt."
  (let ((x1 (first eins))
        (y1 (second eins))  ; zuerst uebersetzen wir mal
        (x2 (first zwei))   ; die Argumente in vernuenftige
        (y2 (second zwei))) ; Variablennamen...
    (if (= x1 x2) (setq x2 (+ x1 .00000000000000000001)))
    (let ((m (/ (- y2 y1) (- x2 x1))))
      (list m (- y1 (* m x1))))))


(defun first_{y≠0} (liste)                                          firstne0
  "firstne0 findet das erste (x y) - Wertepaar, dessen y-Wert groesser
als 0 betraegt. (Voraussetzung: Liste ist stetig wachsend und
geordnet!)"
  (let ((aktuelles (first liste)))
    (cond ((null (rest liste)) 0)
          ((zerop (second aktuelles)) (first_{y≠0} (rest liste)))
          (t aktuelles))))


(defun first_max (liste format)                                     firstmax
  "Die Funktion firstmax findet das erste (x y) - Wertepaar, dessen
y-Wert gleich dem letzten (also maximalen) y-Wert des Profiles
ist. (Voraussetzungen: Liste ist stetig wachsend und geordnet!)"
  (cond ((member format '(:pp :sl))
         (find (cadar (last liste)) liste :key #'cadr :test #'=))
        ((member format '(:lin :linear))
         (list (/ (- 1 (second liste))
                  (first liste))
               1))
        (t (error "Die Funktion firstmax kann nur linear-Parameter oder
Stuetzpunktlisten verarbeiten."))))


(defun stuetzpunkte (profil &key                                    stuetzpunkte
                            (format :pp)
                            (exp_x '(0 1)))
  "Liefert eine Liste aller (notwendigen bei :format :lin) x-Werte
einer Liste von Stuetzpunkten."
  (cond ((member format '(:sl :pp))
         (mapcar #'car profil))
        ((member format '(:lin :linear))
         (cons 0.0 (cond ((= (first profil) 0)
                          (list 1.0))
                         ((minusp (second profil))
```

```
                        (list (coerce (get_{y=0} 0
                                      (second profil)
                                      (first profil))
                                'float)
                              (coerce (get_{y=1} 0
                                      (second profil)
                                      (first profil))
                                'float)))
                       (t (list (coerce (get_{y=1} 0
                                        (second profil)
                                        (first profil))
                                  'float))))))
          ((member format '(:exp :exponential))
           exp_x)))

(defun linear2stuetz (profil)                                          linear2stuetz
  "Liefert fuer die Parameter einer linearen Funktion die Liste aller
notwendigen Stuetzpunkten."
  (cons (list 0.0 (second profil))
        (cond ((= (first profil) 0)
               (list '(1.0 0)))
              ((minusp (second profil))
               (list (list (coerce (get_{y=0} 0
                                   (second profil)
                                   (first profil))
                           'float)
                     0)
                    (list (coerce (get_{y=1} 0
                                   (second profil)
                                   (first profil))
                           'float)
                     1)))
              (t (list (list (coerce (get_{y=1} 0
                                     (second profil)
                                     (first profil))
                               'float)
                         1))))))

(defun exp2stuetz (profil &key (exp_x '(0 1)))                         exp2stuetz
  "Liefert fuer die Parameter einer exponentiellen Funktion die Liste
aller notwendigen Stuetzpunkte."
  (loop for x in exp_x collect
        (list x (expfunkt x profil))))

(defun linearf (Steigung y-Achsenabschnitt)                            linearf
  "Gibt eine Funktion x -> (x*Steigung)+'y-Achsenabschnitt' zurueck."
  (if (minusp Steigung)
      (format t "WARNING (linearf): In performance profiles the gradient is never negative: ~A"
              steigung))
  #'(lambda (x) (+ y-Achsenabschnitt (* x Steigung))))

(defun exponentialf (λ η)                                              exponentialf
  "Gibt eine Funktion x -> 1-eta*e^-lambda*x zurueck."
  (cond ((or (minusp λ) (minusp η))
         (error "Lambda and Eta must be both non-negative: ~A, ~A"
                λ η))
        (t #'(lambda (x)
               (let ((z (- 1 (* η (my-exp (- (* λ x)))))))
                                 (if (minusp z) 0.0 z))))))
```

```
(defun alc (xys)                                                    alc
  "Die Funktion alc konvertiert eine Liste von X-Werten und
Steigungen in eine Liste mit X- und Y-Werten, wie sie von allen
Methoden des Programmes ORCAN akzeptiert werden.

Die Liste, die als Parameter uebergeben wird, muss eine Liste von
einer dreielementigen und einer bis mehreren zweielementigen Listen
sein. Die erste dieser Listen wird als (X-Wert Y-Achsenabschnitt
Steigung) interpretiert, alle anderen als (X-Wert Steigung).

Beispiel:

((x1[y1]s1) (x2s2) (x3s3) (x4s4)) => ((x1y1) (x2y2) (x3y3) (x4y4))
"
  (cond ((null xys) ;;; Kurze Tests, ob Parameter in Ordnung.
         (error "Liste darf nicht leer sein."))
        ((not (listp (car xys)))
         (error "Die Liste muss wiederum Listen enthalten"))
        ((minusp (car (last (car xys))))
         (error "Steigung darf nicht negativ sein: ~A"
                (last (car xys)))))
  (let ((x1 (caar xys)) ;;; Ersteinmal bekommen die Parameter schoene
                        ;;; Namen. Es wird noch abgefangen, ob y
                        ;;; anfangs 0 ist.
        (y1 (if (= (length (car xys)) 2) 0 (cadar xys)))
        (s1 (if (= (length (car xys)) 2) (cadar xys) (car (cddar xys)))))
    (cond ((> y1 1) (error "Y darf nicht > 1 sein: ~A" y1))
          ((= 1 (length xys)) ;;; Wenn nur ein Stuetzpunkt + Steigung da
           (list (list x1 y1) ;;; ist, als lineare Funktion betrachten
                 (list (get_y=1 x1 y1 s1) 1)))
          (t (cons (list x1 y1)
                   (let* ((x2 (caadr xys))
                          (s2 (car (cdadr xys)))
                          (y2 (get_new(y) x1 x2 y1 s1)))
                     ;;; Jetzt sind min. 2 Elemente in der Liste, also
                     ;;; kann man auch den Rest der Parameter benennen,
                     ;;; den naechsten y-Wert berechnen und die ersten
                     ;;; Elemente schon wieder ausgeben.
                     (cond ((> y1 1)
                            (list (list (get_y=1 x1 y1 s1) 1)))
                           ((= 2 (length xys))
                            (list (list x2 y2)
                                  (cond ((zerop s2) (list (1+ x2) y2))
                                        (t (list (get_y=1 x2 y2 s2) 1)))))
                           (t (alc (cons (list x2 y2 s2)
                                         (cddr xys)))))))))))


(defun get_new(y) (x1 x2 y s)                                       getnewy
  "Berechnet den Y-Wert zu X2, wenn zwischen X1 und X2 die Steigung S
herrscht und der Y-Wert von X1 Y ist."
  (+ (* s (- x2 x1)) y))


(defun get_y=1 (x1 y s)                                             getyeq1
  "Berechnet den X-Wert, bei dem Y=1 ist, wenn zwischen X1 und X2 die
Steigung S herrscht und der Y-Wert von X1 Y ist."
  (+ (/ (- 1 y) s) x1))


(defun get_y=0 (x1 y s)                                             getyeq0
```

```
  "Berechnet den X-Wert, bei dem Y=0 ist, wenn zwischen X1 und X2 die
Steigung S herrscht und der Y-Wert von X1 Y ist."
  (+ (/ (- y) s) x1))


(defun my-exp (num)                                                    my-exp
  "my-exp gleicht folgenden Lucid-Bug aus:
210
ORCAN> (exp -708)
3.307553003638408E-308
; Arg klein, nicht? :-)
ORCAN> (exp -709)
>>Error: A condition of type FLOATING-POINT-UNDERFLOW occurred.

Die Funktion my-exp ersetzt ueberall das normale exp. Fuer besonders
kleine Werte approximieren wir einfach Null, fuer besonders grosse
unendlich... So einfach geht das. Ausserdem: Ich kann mir nicht
220 vorstellen, dass Lucid Common Lisp viel genauer rechnet als ich
approximiere... (SCNR. Axel)"

  #--Lucid (exp num)
  #+Lucid (cond ((< num -708) 0.0)
                ((> num 709) SYSTEM:∞)
                (t (exp num))))


(defun +* (&rest l)                                                    +*
  (let ((l2 (apply '* l)))
230   (if (not (plusp l2))
        (+ (apply '+ l) l2)
      l2)))
```

D. Verzeichnisse

D.1. Abbildungen

D.2. Algorithmen

D.3. Tabellen

D.4. Stichworte

Sonderzeichen

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

V

W

Z

D.5. Literatur

ACL. [1996]. *Allegro Common Lisp User Guide.*

Adobe Systems Inc. (Hrsg.). [1985]. *PostScript® Language Tutorial and Cookbook.* Reading, Massachusetts: Addison Wesley Publishing Company, Inc. (ISBN: 0-201-10179-3)

Adobe Systems Inc. (Hrsg.). [1991a]. *PostScript® - Einführung und Leitfaden (Deutsche Übersetzung).* Bonn: Addison Wesley (Deutschland) GmbH. (ISBN: 3-89319-070-5)

Adobe Systems Inc. (Hrsg.). [1991b]. *PostScript® Language Reference Manual* (2. Auflage). Reading, Massachusetts: Addison Wesley Publishing Company Inc. (ISBN: 0-201-18127-4)

Baus, A. & Beckert, A. [1998]. ORCAN - *Implementation von Methoden zur Compilierung von Anytime Algorithmen.* Saarbrücken: Universität des Saarlandes, Fachbereich 14 - Informatik, Lehrstuhl für Künstliche Intelligenz Prof. Dr. Dr. h.c. Wahlster. (Fortgeschrittenenpraktikum am Lehrstuhl für Künstliche Intelligenz)

Beckert, A. [2001]. ORCAN*: Laufzeitmessungen von Methoden zur Anytime-Kompilierung* (Memo Nr. 72). SFB378.

Blocher, A. [1999]. *Ressourcenadaptierende Raumbeschreibung: Ein beschränkt-optimaler Lokalisationsagent.* Doktorarbeit, Fachbereich Informatik, Technische Fakultät, Universität des Saarlandes, Saarbrücken.

Bronstein, I. N. & Semendjajew, K. A. [1967]. *Taschenbuch der Mathematik* (7. Auflage). Zürich, Frankfurt am Main: Harri Deutsch Verlag.

Church, A. [1941]. „The Calculi of Lambda-Conversion." In *Annals of Mathematical Studies* (Vol. 6). Princeton, New Jersey: Princeton University Press. (Reprinted by Klaus Reprints, New York, 1965)

Dean, T. & Boddy, M. [1988]. „An Analysis of Time-Dependent Planning." In *Proceedings of the Seventh National Conference on Artificial Intelligence* (Seite 49-54). Minneapolis, Minnesota.

Die Duden-Redaktion (Hrsg.). [1969]. *Duden: »Rechnen und Mathematik«* (3. Auflage). Mannheim, Wien, Zürich: Bibliographisches Institut.

Die Duden-Redaktion (Hrsg.). [1986]. *Duden, Band 2: »Sinn- und sachverwandte Wörter«* (2. Auflage). Mannheim, Wien, Zürich: Bibliographisches Institut.

Die Duden-Redaktion (Hrsg.). [1996]. *Duden, Band 1: »Rechtschreibung der deutschen Sprache und Fremdwörter«* (19. Auflage). Mannheim, Wien, Zürich: Bibliographisches Institut.

Dorigo, M. & Di Caro, G. [1996]. „Heuristics from Nature for Hard Combinatorial Optimization Problems." In *International Transactions in Operational Research* (Vols. 3, 1, Seite 1-21).

Dorigo, M. & Gambardella, L. M. [1998]. *Ant Algorithms for Discrete Optimization* (Technical Report Nr. TR/IRIDIA/1998-10). Bruxelles, Belgium: IRIDIA, Université Libre de Bruxelles.

Gapp, K.-P. [1997]. *Objektlokalisation: Ein System zur sprachlichen Raumbeschreibung.* Wiesbaden: Deutscher Universitätsverlag.

Goldberg, D. [1989]. *Genetic Algorithms in Search, Optimization and Machine Learning.* Reading, Massachusetts: Addison Wesley.

Görz, G. & Kesseler, M. [1994]. „Anytime Algorithms for Speech Parsing?" In *Proc. COLING-94* (Seite 997-1001). Kyoto.

Grass, J. [1996]. „Reasoning about computational resource allocation - An introduction to anytime algorithms." *ACM Crossroads.*

Horvitz, E. J. [1987]. „Reasoning about Beliefs and Actions under Computational Resource Constraints." In *Proc. of the Third Workshop on Uncertainty in Artificial Intelligence, Seattle WA* (Seite 429-444). Mountain View, CA.

HP-Lucid. [1990, jun]. *Lucid Common Lisp/HP Delivery Tool Kit.*

Jameson, A. & Buchholz, K. [1998]. „Einleitung zum Themenheft „Ressourcenadaptive kognitive Prozesse"." *Kognitionswissenschaft, 7*(3), 95-100.

LIQUID. [1997]. *Liquid Common Lisp documentation.* URL: http://www.harlequin.com/education/books/lcl_doc.html.

Maaß, W. [1996]. *Von visuellen Daten zu inkrementellen Wegbeschreibungen in dreidimensionalen Umgebungen: Das Modell eines kognitiven Agenten.* Doktorarbeit, Fachbereich Informatik, Technische Fakultät, Universität des Saarlandes, Saarbrücken.

Murty, K. G. [1995]. *Operations Research: Deterministic Optimization Models.* Englewood Cliffs, New Jersey: Prentice Hall. (ISBN: 0-13-056517-2)

PERL. [2000]. *PERL - Practical Extraction and Report Language.* URL: http://www.perl.com/.

REAL. [1996]. *REAL - Ressourcenadaptive Lokalisation.* URL: http://w5.cs.uni-sb.de/real.html.

Rechenberg, I. [1972]. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Stuttgart: Frommann-Holzboog.

Russell, S. J. & Subramanian, D. [1995]. „Provably bounded-optimal agents." *Journal of Artificial Intelligence Research, 1*(1), 1-36.

Russell, S. J., Subramanian, D. & Parr, R. [1993]. „Provably bounded-optimal agents." In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93).*

Russell, S. J. & Zilberstein, S. [1996]. „Optimal composition of real-time systems." *Artificial Intelligence, 1-2*(82), 181-213.

Schwefel, H. P. [1977]. *Numerische Optimierung von Computermodellen mittels der Evolutionsstrategie.* Basel und Stuttgart: Birkhäuser Verlag.

SFB. [1997]. *Sonderforschungsbereich 378 „Ressourcenadaptive kognitive Prozesse".* URL: http://www.coli.uni-sb.de/sfb378/.

Steele, G. L. [1994]. *Common Lisp: The Language* (2. Auflage). Bedford, Massachusetts: Digital Press. (ISBN: 1-55558-041-6)

Stopp, E. [1998]. *Natürlichsprachliche Dialoge mit mobilen Robotern auf der Basis einer ressourcenadaptiven Referenzsemantik räumlicher Umgebungen.* Doktorarbeit, Fachbereich Informatik, Technische Fakultät, Universität des Saarlandes, Saarbrücken.

Swain, R. [1999]. *PERL 5 Short Reference.*

Various Artists. [1999]. *The LGrind package.*

Wahlster, W. [2000]. *Folien zur Vorlesung* Einführung in die Künstliche Intelligenz *im Sommersemester 2000 an der Universität des Saarlandes.*

Wahlster, W., Blocher, A., Baus, J., Stopp, E. & Speiser, H. [1998]. „Ressourcenadaptierende Objektlokalisation: Sprachliche Raumbeschreibung unter Zeitdruck." *Kognitionswissenschaft, Sonderheft zum Sonderforschungsbereich 378, 7*(3).

Wahlster, W. & Tack, W. [1997]. „SFB 378: Ressourcenadaptive Kognitive Prozesse." In M. Jarke, K. Pasedach & K. Pohl (Hrsg.), *Informatik'97 - Informatik als Innovationsmotor, 27. Jahrestagung der Gesellschaft für Informatik, Aachen, 24.-26. September 1997* (Seite 51-57). Berlin, Heidelberg: Springer.

Wittig, F. [1998]. *JAMES: Java Anytime Management and Editor System.* Diplomarbeit, Universität des Saarlandes, Technische Fakultät, Saarbrücken.

Zilberstein, S. [1993]. *Operational Rationality through Compilation of Anytime Algorithms.* Doktorarbeit, University of California at Berkeley, Berkeley.