universität wien

MAGISTERARBEIT / MASTER'S THESIS

Titel der Magisterarbeit / Title of the Master's Thesis

„Heuristische Optimierung des multi-modalen RCPSP mit stochastischen Bearbeitungsdauern“

verfasst von / submitted by

Mag. Bernhard Hrobath, Bakk.

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Magister der Sozial- und Wirtschaftswissenschaften (Mag. rer. soc. oec.)

Wien, 2017 / Vienna 2017

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 066 951
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Magisterstudium Statistik UG2002
Betreut von / Supervisor:	Univ.-Prof. Mag. Dr. Walter Gutjahr

Eidesstattliche Erklärung

Ich erkläre eidesstattlich, dass ich die Arbeit selbständig angefertigt, keine anderen als die angegebenen Hilfsmittel benutzt und alle aus ungedruckten Quellen, gedruckter Literatur oder aus dem Internet im Wortlaut oder im wesentlichen Inhalt übernommenen Formulierungen und Konzepte gemäß den Richtlinien wissenschaftlicher Arbeiten zitiert, durch Fußnoten gekennzeichnet bzw. mit genauer Quellenangabe kenntlich gemacht habe. Diese schriftliche Arbeit wurde noch an keiner anderen Stelle vorgelegt.

Wien, 2017

Bernhard Hrobath

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich bei der Erstellung dieser Arbeit und während der Zeit meines Studiums unterstützt haben.

Allen voran gebührt die größte Anerkennung meiner Frau Nicole. Ohne ihre Anstrengungen, mir in der Zeit der Fertigstellung dieser Arbeit sämtliche anderen Pflichten abzunehmen, wäre das Verfassen ungleich langwieriger und schwieriger gewesen.

Bei meinem Betreuer Prof. Gutjahr möchte ich mich für seine Geduld, seine Unterstützung und die konstruktiven Gespräche während der Erstellung der Arbeit bedanken.

Meiner Familie danke ich für die Unterstützung während meiner Studienzeit und in allen anderen Phasen meines Lebens.

Inhaltsverzeichnis

1 Einleitung

Beim *Resource-Constrained Project Scheduling Problem* (RCPSP) handelt es sich um ein kombinatorisches Optimierungsproblem. Prinzipiell sollen Startzeiten für Teilaktivitäten eines Projekts mit unterschiedlichen Bearbeitungsdauern gefunden werden. Dabei müssen Vorgängerbeziehungen zwischen den Aktivitäten sowie Ressourcenverbrauch und -verfügbarkeit berücksichtigt werden. Im Sinne eines Optimierungsproblems soll eine Zielfunktion (z.B. die *makespan*, d.h. die gesamte Bearbeitungsdauer des Projekts) optimiert werden.

Schon in der klassischen Formulierung ist das RCPSP ein schwieriges Optimierungsproblem. Es handelt sich um ein NP-schweres Problem (Blasewicz, Lenstra & Rinnoy Kan, 1983). Das RCPSP ist bereits seit mehreren Jahrzehnten Thema vieler Forschungsarbeiten. Überblicksarbeiten zu unterschiedlichen Bereichen findet man beispielsweise bei Özdamar und Ulusoy (1995), Bruckner, Drexl, Möhrung, Neumann und Pesch (1999), Kolisch und Padman (2001) oder Hartmann und Briskorn (2010).

Es existieren diverse Erweiterungen der Problemformulierung. Der multi-modale Fall liegt vor, wenn die Aktivitäten in mehreren Ausführungsmodi durchgeführt werden können. Eine weitere Verallgemeinerung betrifft die Bearbeitungsdauern. Diese können statt über eine deterministische Modellierung alternativ als Zufallszeiten, d.h. stochastisch modelliert werden. Diese Arbeit behandelt eben diese beiden Erweiterungen gleichzeitig, d.h. es soll das stochastische multi-modale RCPSP betrachtet werden.

1.1 Problemformulierung

Zunächst wird die Problemformulierung in Anlehnung an Gutjahr (2015) schrittweise aufgebaut. Beginnend mit dem klassischen RCPSP werden anschließend die in dieser Arbeit verwendeten Erweiterungen beschrieben.

1.1.1 Das klassische RCPSP

Die klassische Version des (deterministischen) RCPSP lässt sich folgendermaßen beschreiben. Die Notation und Formulierung folgt dabei weitgehend der von Hartmann und Briskorn (2010), Ballestín und Leus (2009) und Gutjahr (2015).

Ein Projekt besteht aus J Aktivitäten (Jobs). Um das Projekt abzuschließen, müssen alle Aktivitäten abgeschlossen sein. Eine gängige Formulierung beinhaltet zwei zusätzliche

Dummy-Aktivitäten mit den Indizes 0 und $J+1$. Dabei handelt es sich um den Projektstart sowie das Projektende. Im Folgenden wird die Indizierung $j = 0, 1, \dots, n, J+1$ für die Kennzeichnung der Aktivitäten verwendet.

Jeder Aktivität j ist eine (deterministische) Bearbeitungsdauer d_j zugeordnet. Für die beiden Dummy-Aktivitäten wird $d_0 = d_{J+1} = 0$ angenommen.

Für das Projekt müssen K erneuerbare Ressourcen berücksichtigt werden. Jede Aktivität j verbraucht während der Bearbeitung r_{jk} Einheiten der k-ten erneuerbaren Ressource, $k = 1, \dots, K$. Sobald die Aktivität beendet ist, werden diese Mengen der Ressourcen wieder frei. Im Projekt sind dabei Kapazitäten $a_1, \dots, a_K$ für die erneuerbaren Ressourcen vorgegeben. Diese dürfen zu keinem Zeitpunkt überschritten werden. Die Dummy-Aktivitäten verbrauchen keinerlei Ressourcen, d.h. $r_{0k} = r_{J+1k} = 0$ für $k = 1, \dots, K$.

Die Aktivitäten stehen in bestimmten Vorgängerbeziehungen zueinander, welche über eine Menge von Paaren $H = \{(i,j)\}$ beschrieben werden können. Falls $(i,j) \in H$, dann ist i Vorgänger von j, d.h. Aktivität i muss beendet sein, bevor Aktivität j starten kann. Für die Dummy-Aktivität 0 gilt dabei, dass sie Vorgänger aller anderen Aktivitäten ist. Die Dummy-Aktivität $J+1$ ist Nachfolger aller anderen Aktivitäten, d.h. jede Aktivität ist Vorgänger der Dummy-Aktivität $J+1$.

Für die Menge H gelten mindestens folgende Einschränkungen, damit die Lösungsmenge überhaupt zulässige Lösungen enthält:

- $(i,i) \notin H$, für alle $i = 0, \dots, n+1$.
- Wenn $(i,j) \in H$, dann $(j,i) \notin H$.
- Wenn $(i,j) \in H$ und $(j,l) \in H$, dann $(i,l) \in H$.

Sparsamer lassen sich die Vorgängerbeziehungen über ein gerichtetes, azyklisches Netzwerk darstellen, wobei die Knoten die Aktivitäten darstellen, und Kanten zwischen zwei Knoten eine Vorgängerbeziehung repräsentieren (Hartmann & Briskorn, 2010). Somit müssen zur vollständigen Beschreibung nur die direkten Vorgängerbeziehungen berücksichtigt werden.

Die Aktivitäten werden zu den Zeitpunkten $S_0 = 0, S_1, \dots, S_i, \dots, S_J, S_{J+1}$ gestartet und müssen ohne Unterbrechung beendet werden. Eine zulässige Lösung S ist eine Menge von Startzeitpunkten für jede Aktivität (einem *schedule*), sodass die Vorgängerbeziehungen eingehalten und Ressourceneinschränkungen nicht verletzt werden.

Die Zielgröße ist dabei die Gesamtbearbeitungsdauer des Projekts – die sogenannte *makespan*. In der hier angegebenen Formulierung, entspricht diese der S_{J+1}, also der Startzeit

der letzten Dummy-Aktivität. (Diese entspricht wegen $d_{J+1} = 0$ auch deren Beendigungszeit und damit wegen der Vorgängerbeziehungen aller anderen Aktivitäten zur Aktivität $J + 1$ auch dem Zeitpunkt der Komplettierung des gesamten Projekts. Wegen $S_0 = 0$ entspricht dies letztendlich auch der Dauer der Bearbeitungszeit – der *makespan*).

Unter der Annahme, dass die *makespan* minimiert werden soll, lassen sich diese Formulierungen und Bedingungen folgendermaßen formal beschreiben:

$$\min \quad S_{J+1}$$

s.t.

$$S_0 = 0$$

$$d_0 = d_{J+1} = 0$$

$$S_j \geq S_i + d_i, \qquad \forall (i,j) \in H$$

$$x_{jt} := \mathbb{1}_{\{S_j \leq t \leq S_j + d_j\}}$$ (Indikator, ob Aktivität j zum Zeitpunkt t in Bearbeitung ist)

$$\sum_{j=0}^{J+1} r_{jk} x_{jt} \leq a_k, \qquad \forall k = 1, \dots, K, \forall t \leq S_{J+1}$$

1.1.2 Erweiterung: Multi-modales RCPSP

Die oben beschriebene klassische Problemformulierung kann durch die Annahme mehrerer Ausführungsmodi der Aktivitäten verallgemeinert werden.

Dabei kann jede Aktivität j in M_j Modi ausgeführt werden. Die verbrauchten Ressourcenkapazitäten können nun innerhalb einer Aktivität von Modus zu Modus unterschiedlich sein und werden mit einem zusätzlichen Index versehen. Somit verbraucht jede Aktivität j im Modus m nun r_{jmk} Einheiten der Ressource k. Die Bearbeitungsdauern hängen ebenfalls vom Modus ab und werden nun mit d_{jm} bezeichnet.

Eine Lösung muss nun zusätzlich zu den Startzeiten jeder Aktivität j genau einen möglichen Modus $m_j \in \{1, \dots, M_j\}$ zuweisen. Die formale Repräsentation des Problems kann dann folgendermaßen erweitert beschrieben werden:

$\min \quad S_{J+1}$

s.t.

$S_0 = 0$

$d_0 = d_{J+1} = 0$

$S_j \geq S_i + d_i, \qquad \forall (i,j) \in H$

$y_{jm} := \mathbb{1}_{\{m_j = m\}}$ (Indikator, ob Aktivität j im Modus m ausgeführt wird)

$$\sum_{m=1}^{M_j} y_{jm} = 1, \qquad \forall j = 0,1, \dots, J+1$$

$x_{jmt} := \mathbb{1}_{\{S_j \leq t \leq S_j + d_{jm}\}} y_{jm}$ (Indikator, ob Aktivität j zum Zeitpunkt t in Bearbeitung ist; abhängig wegen y_{jm} auch vom Ausführungsmodus m)

$$\sum_{j=0}^{J+1} \sum_{m=1}^{M_j} r_{jmk} x_{jmt} \leq a_k\,, \qquad \forall k = 1, \dots, K, \forall t \leq S_{J+1}$$

1.1.3 Erweiterung: Nicht erneuerbare Ressourcen

Die multi-modale Version des RCPSP (im weiteren MMRCPSP) dient nun als Ausgangsbasis für die nächste Erweiterung. Zusätzlich zu den erneuerbaren Ressourcen, welche zu jedem Zeitpunkt in einer bestimmten Kapazität zur Verfügung stehen, werden auch N nicht-erneuerbare Ressourcen mitberücksichtigt. Diese stehen für das gesamte Projekt in den Kapazitäten $b_1, \dots, b_N$ zur Verfügung. Jede Aktivität j verbraucht im Modus m dabei einmalig e_{jmn} Einheiten der n-ten Ressource. Die Annahme ist nun, dass folgende Bedingung zusätzlich zu den oben angegebenen Bedingungen/Definitionen des MMRCPSP gilt:

$$\sum_{j=0}^{J+1} \sum_{m=1}^{M_j} e_{jmn} y_{jm} \leq b_n\,, \qquad \forall n = 1, \dots, N$$

Dies bedeutet, die von allen Aktivitäten (im jeweiligen Ausführungsmodus) verbrauchte Menge der Ressource n, darf die Gesamtkapazität dieser Ressource nicht überschreiten. Die benötigte Ressourcenmenge hängt innerhalb der Aktivitäten jedoch auch nur von der Auswahl der Modi ab. Es ergibt sich somit eine Einschränkung der zulässigen Lösungen, indem bestimmte Kombinationen an Modi von vornherein ausgeschlossen werden. Auf die

Zielfunktion (die *makespan*) haben die nicht-erneuerbaren Ressourcen somit nur indirekt einen Einfluss, indem bestimmte Zusammensetzungen der Modi der Aktivitäten nicht zulässig sind.

An dieser Stelle sei angemerkt, dass in der vorliegenden Arbeit ein anderer Zugang der Berücksichtigung der nicht-erneuerbaren Ressourcen gewählt wird. Statt einer harten Einschränkung der Menge an Lösungen, wird eine Überschreitung der Ressourcenkapazität als Strafterm in die Zielfunktion integriert. Dabei kommt es zu einer Verschlechterung der Zielfunktion, falls zu viele Ressourcen verbraucht werden. Es bleiben jedoch alle Lösungen (in Bezug auf die Auswahl der Modi) prinzipiell erhalten. Das genaue Vorgehen wird an späterer Stelle beschrieben.

1.1.4 Erweiterung: Stochastische Bearbeitungsdauern

In den oben angegebenen Problemrepräsentationen sind die Bearbeitungsdauern deterministisch angenommen. D.h. die je nach Auswahl der Modi dauert die Bearbeitung immer eine fixe, als bekannt angenommene Zeit. Die letzte Erweiterung, die letztendlich zur in dieser Arbeit verwendeten Problem-Variante führt, ist die Modellierung der Bearbeitungszeiten als Zufallsvariablen. Es wird dabei angenommen, dass die Bearbeitungsdauer der Aktivität j im Modus m aus einer parametrischen Verteilung $F(\theta_{jm})$ stammt, d.h. $d_{jm} \sim F(\theta_{jm})$. Der Parametervektor θ_{jm} hängt unter Umständen von j und m ab, wird jedoch als bekannt und deterministisch angenommen. Die für diese Arbeit gewählten Modellierung der Verteilungen, sowie die Konstruktion der Parameter wird im Abschnitt 1.2 beschrieben.

Die Wahl von stochastischen Bearbeitungsdauern führt dazu, dass einige der oben beschriebenen Aspekte der Problemstellung nicht mehr sinnvoll sind. Diese Problemstellung erfordert eine neue Repräsentation einer Lösung, sowie eine Umformulierung der Zielfunktion.

1.1.5 Adaptierte Lösungsrepräsentation

Die Startzeiten der Aktivitäten (in Kombination mit dem Modus der Aktivitäten) können nicht mehr sinnvoll als Repräsentation einer Lösung verwendet werden. Dies liegt daran, dass zu jedem Zeitpunkt t nur die Information verwendet werden kann, die bisher bekannt ist (Ballestín, 2007). Zu jedem Zeitpunkt t ist neben den grundsätzlich als bekannt angenommenen Parametern der Zufallsverteilungen nur bekannt, welche Aktivitäten bisher beendet wurden, bzw. gerade noch aktiv sind. Somit erfordert diese Problemformulierung eine andere Lösungsrepräsentation, welche sich auf diese Informationen stützt.

Grundlage dafür bildet eine Politik (*policy*), die Regeln vorgibt, nach denen bestimmte Aktionen zu den Entscheidungszeitpunkten gewählt werden (Ballestín, 2007). Entscheidungszeitpunkte sind dabei neben dem Projektstart $t_0 = 0$ die Zeitpunkte, an denen mindestens eine gerade noch laufende Aktivität beendet wird (Ballestín & Leus, 2009). Zu beachten ist, dass die Entscheidungszeitpunkte aufgrund der zufälligen Bearbeitungszeiten der Aktivitäten ebenfalls zufallsabhängig sind. Insgesamt kann es neben t_0 maximal J solche Entscheidungszeitpunkte geben. Eine Aktion besteht in der Entscheidung, ob und welche der noch nicht gestarteten Aktivitäten gestartet werden (Ballestín, 2007). Für die Entscheidung (Auswahl der zu startenden Aktivitäten) dürfen nur die zum Entscheidungszeitpunkt bekannten Informationen genutzt werden (Ballestín, 2007).

Übliche Politiken für das RCPSP sind Prioritätslisten (Ballestín & Leus, 2009). Dabei handelt es sich um eine Permutation der Aktivitäten, wobei die Priorität durch die Position in der Liste gegeben ist. Eine frühere Position geht mit einer höheren Priorität einher. Die Priorität bestimmt die Reihenfolge, in der die Aktivitäten zu einem Entscheidungszeitpunkt gestartet werden. Im Folgenden wird die hier verwendete Prioritätspolitik – *priority policy* (Ballestín, 2007) beschrieben. Diese Politik verwendet die Prioritätsliste der Aktivitäten insofern, als dass zu jedem Entscheidungszeitpunkt t alle möglichen Aktivitäten gestartet werden, die die Ressourceneinschränkungen nicht verletzen, sowie die Vorgängerbeziehungen einhalten. Dabei werden die Entscheidungen für den Start der Aktivitäten in der Reihenfolge der Prioritätsliste abgearbeitet. Diese Bearbeitungsweise entspricht dem *parallel schedule generation scheme (parallel SGS)* (Kolisch & Hartmann, 1998; Ballestín, 2007) nach einer Prioritätsregel.

Als Lösungsrepräsentation ergibt sich dann gemeinsam mit der multi-modalen Problemstellung eine Lösung nun als Paar einer Prioritätsliste P sowie einer Auswahl M an Modi $m_j \in \{1, \dots, M_j\}$ für jede Aktivität j.

1.1.6 Neue Zielfunktion

Die Formulierung der Bearbeitungszeiten als Zufallsvariablen sowie die Repräsentation einer Lösung nach obiger Beschreibung führt dazu, dass der Beendigungszeitpunkt des Projekts $C_{max} = S_{J+1}$ nun ebenfalls eine zufallsabhängige Größe darstellt. Daher ist die Formulierung einer darauf angepassten Zielfunktion nötig. Eine übliche Zielfunktion stellt nun beispielsweise der Erwartungswert der Gesamtbearbeitungsdauer $\mathbb{E}(C_{max})$ dar (Ballestín, 2007). In der vorliegenden Arbeit wird jedoch ein neuer Ansatz gewählt, welcher eher einer risiko-aversen Strategie entspricht. Es soll vermieden werden, dass übermäßig lange Projektdauern auftreten. Daher wird als Zielgröße der *average value at risk* zum Niveau $\alpha \in (0, 1)$, kurz

$AVaR_\alpha$, gewählt. Dieser ist hier für eine zufällige Größe $X \sim F(X)$ und unter Annahme einer streng monotonen Verteilungsfunktion F definiert als

$$AVaR_\alpha(X) = \mathbb{E}\big(X | X > \mathrm{F}^{-1}(1-\alpha)\big).$$

Für diese Definition wird angenommen, dass hohe Werte von X einen hohen Verlust darstellen. Dies ist für die vorliegende Problemstellung sinnvoll, da hohe Bearbeitungszeiten einen Verlust darstellen. Es soll weiterhin ein Minimierungsproblem betrachtet werden, wobei äquivalent zur Minimierung der *makespan* C_{max} bzw. des Erwartungswerts der *makespan* $\mathbb{E}(C_{max})$, hier eine Minimierung des $AVaR_\alpha$ der *makespan*, also $AVaR_\alpha(C_{max})$ angestrebt werden soll.

Zu beachten ist, dass für die Bestimmung der Zielfunktion die Verteilung von C_{max} unter Verwendung der bekannten Informationen bestimmt werden müsste. Eine Alternative Vorgehensweise besteht darin, $AVaR_\alpha(C_{max})$ auf Basis von Szenarien zu schätzen (Ballestín, 2007).

Um die Zielfunktion für eine bestimmte Lösung S zu schätzen, wird im Vorhinein eine bestimmte Anzahl N_{scen} an Szenarien simuliert. Dabei werden die Bearbeitungszeiten aus den vorgegebenen Verteilungen zufällig und unabhängig voneinander gezogen. Diese Szenarien werden als deterministische Problemstellungen betrachtet, für welche jeweils die *makespan* unter Verwendung der Lösung S (d.h. dem Paar von Prioritätsliste und Modusliste) und der gewählten Politik (*parallel SGS* mit Prioritätsregel) bestimmt wird. Somit erhält man eine Stichprobe von *makespans* $C_1, \dots, C_{N_{scen}}$. Eine Schätzung der Zielfunktion ergibt sich nun aus der Schätzung des $AVaR_\alpha$ durch folgende Vorgehensweise:

1.) Ordne die Stichprobe der *makespans*: $C_{(1)}, \dots, C_{(N_{scen})}$
2.) Bestimme $u = \min\{x \in \mathbb{N}, x \leq (1-\alpha) \cdot N_{scen}\} + 1$
3.) Schätze $AVaR_\alpha(C_{max})$ durch

$$\widehat{AVaR}_\alpha(C_{max}) = \frac{1}{N_{scen} - \mathrm{u} + 1} \sum_{i=u}^{N_{scen}} C_{(i)}$$

Somit wird der $AVaR_\alpha$ durch den Mittelwert der Beobachtungen geschätzt, die größer als das empirische $(1-\alpha)$-Quantil sind. Falls $(1-\alpha) \cdot N_{scen} \in \mathbb{N}$ handelt es sich um den Mittelwert der $\alpha \cdot 100$ % größten Beobachtungen.

Diese Schätzung des $AVaR_\alpha$ für eine bestimmte Lösung wird als Grundlage der zu optimierenden Zielfunktion verwendet. Die Zielfunktion berücksichtigt eine weitere

Komponente der Probleminstanzen – die nicht-erneuerbaren Ressourcen. Diese Ressourcen stehen für das gesamte Projekt in einer vorgegebenen Menge zur Verfügung. Jede Aktivität verbraucht im jeweiligen Ausführungsmodus eine bestimmte Menge der Ressourcen. Somit kann es sich ergeben, dass die in einer Lösung verwendeten Modi dazu führen, dass die Aktivitäten insgesamt mehr Einheiten einer Ressource verbrauchen würden, als verfügbar ist. Eine Option wäre, diese Lösungen als nicht zulässig zu deklarieren, und auszuscheiden. Dies führt jedoch unter Umständen zu einer drastischen Reduktion der Menge an zulässigen Lösungen. Darum wird hier ein anderer Ansatz gewählt. Eine weitere Möglichkeit ist es, die Überschreitung der Ressourcenkapazität als Pönalisierung in die Zielfunktion einzubauen und unzulässige Lösungen im Sinne einer Erhöhung der Zielfunktion zu bestrafen. Zur Bemessung der Überschreitung der Kapazität für die nicht-erneuerbaren Ressource n definieren wir

$$h_n(S) = \max\left\{0, \sum_{j=0}^{J+1} \sum_{m=1}^{M_j} e_{jmn} y_{jm}(S) - b_n\right\}$$

S ist dabei die zu evaluierende Lösung, b_n die Kapazität der n-ten nicht erneuerbaren Ressource. $y_{jm}(S)$ entspricht dabei dem Indikator, ob in der Lösung S die Aktivität j im Modus m ausgeführt wird.

Die folgende Wahl der Zielfunktion realisiert dann den Einbezug der Überschreitungen der Ressourcenkapazitäten:

$$g(S) = \widehat{AVaR}_\alpha\big(C_{max}(S)\big) + \sum_{n=1}^{N} c_n \cdot h_n(y)$$

S repräsentiert wieder eine Lösung und $C_{max}(S)$ entspricht dem für die Szenarien berechneten *makespans*, wenn die in S codierte Prioritätsliste der Aktivitäten und die entsprechenden Modi verwendet werden. Die Konstanten c_n können für eine eventuelle unterschiedlich gewünschte Gewichtung bzw. für das Ausmaß der Bestrafung pro Einheit der Überschreitung gewählt werden. Wird ein $c_n = 0$ gesetzt, wird die entsprechende Ressource n ignoriert.

Klarerweise entspricht die Zielfunktion für eine im Sinne der nicht-erneuerbaren Ressourcen zulässige Lösung einfach dem geschätzten $AVaR_\alpha$, da der Strafterm dann nicht schlagend wird.

Somit können prinzipiell alle möglichen Lösungen evaluiert werden, und es muss nicht im Vorhinein die Menge an zulässigen Lösungen bestimmt werden. Dies ist insofern verteilhaft, da nach Kolisch und Drexl (1997) bereits das Auffinden einer im Sinne der nicht-erneuerbaren Ressourcen zulässigen Lösung im MMRCPSP ein NP-vollständiges Problem darstellt, falls $N \geq 2$ und $M_j \geq 2, j = 1, \dots, J$. Diese Formulierung der Zielfunktion hat einen weiteren Vorteil,

welcher später in den Algorithmen Anwendung findet. Die Wahl der Konstanten $c_1, \dots, c_N$ kann adaptiv erfolgen. Zu Beginn des Algorithmus werden die Konstanten auf null gesetzt, oder sehr niedrig gewählt. Dann kann der gesamte Lösungsraum ohne Einschränkungen abgesucht werden. Schrittweise werden diese dann erhöht, sodass gegen Ende der Laufzeit praktisch nur mehr zulässige Lösungen eine Chance haben, als derzeit beste Lösung in Frage zu kommen. Eine weitere Idee dieses Vorgehens ist, dass möglicherweise Lösungen in der Umgebung einer im Sinne des $AVaR_\alpha$ guten, jedoch unzulässigen Lösung ähnlich gute aber zulässig Lösungen zu finden sind. Da beide Algorithmen kleine Änderungen der Lösungen verwenden, besteht daher die Möglichkeit, gute zulässige Lösungen zu finden, indem gute unzulässige Lösungen schrittweise verbessert werden.

1.1.7 Vereinfachung der Lösungsrepräsentation

Die multi-modale Problemstellung erfordert die oben angegeben die Repräsentation einer Lösung durch zwei Teile: eine Prioritätsliste P der Aktivitäten und einer Modusliste M, welche den je Aktivität verwendeten Modus angibt. Im folgenden Abschnitt werden einige zusätzliche Überlegungen zu den Eigenschaften der so repräsentierten Lösungen behandelt.

Die Prioritätsliste P entspricht einer Permutation der Indizes der Aktivitäten. Der Einfachheit halber wird bei der Indizierung von der oben verwendeten Form $0, 1, \dots, J+1$ abgewichen. Die Indizes der Aktivitäten werden verschoben, und beginnen ab nun bei 1 (der Dummy-Aktivität des Projektstarts) und enden bei $J+2$ (der Dummy-Aktivität des Projektendes) gewählt. Somit entspricht eine Prioritätsliste einer Permutation von $\{1, 2, \dots, J+1, J+2\}$.

Die Liste der Modi ist dann ein geordneter Vektor mit genau $J+2$ Einträgen. Dabei entspricht der j-te Eintrag demjenigen Modus, der für die Aktivität mit dem Index j vorgesehen ist. Somit entspricht eine Liste von Modi einem Vektor $(m_1, m_2, \dots, m_{J+1}, m_{J+2})$, wobei $m_j \in \{1, \dots, M_j\}$ für alle $j = 1, \dots, J+2$ gelten muss.

Ein Beispiel für eine Lösungsrepräsentation bei einer Instanz mit $J = 10$ Aktivitäten und jeweils drei Modi pro Aktivität (ein Modus für die Dummy-Aktivitäten 1 und $J+2$) wäre also:

Prioritätsliste: $P = (9, 3, 10, 1, 7, 12, 5, 11, 6, 4, 8, 2)$

Modusliste: $M = (1, 2, 2, 3, 3, 3, 1, 3, 3, 3, 1, 1)$

Wichtig ist, dass eine Lösung nur als Paar eben dieser beiden Listen betrachtet werden kann.

1.1.8 Vorgänger-Zulässigkeit von Lösungen

Aufgrund der Vorgängerbeziehungen zwischen den Aktivitäten, können verschiedene Prioritätslisten (bei fixer Modusliste) in Bezug auf die Zielfunktion als äquivalent betrachtet werden. Angenommen eine Aktivität j ist eine Vorgängeraktivität einer anderen Aktivität i. Dann kann i erst dann gestartet werden, wenn j abgeschlossen ist. Im *parallel SGS* wird die Aktivität i daher an den Entscheidungszeitpunkten auf jeden Fall so lange übersprungen, bis Aktivität j abgeschlossen ist. Jede Prioritätsliste (bei fixer Modusliste) in der Aktivität j nach i kommt, kann daher so verändert werden, dass i an die Stelle unmittelbar nach j verschoben wird, ohne die Reihenfolge der anderen Aktivitäten zu verändern. Die so erhaltene Lösung führt dann zwangsläufig zum gleichen *schedule* und damit zur gleichen *makespan* wie die ursprüngliche Lösung. Eine Lösung in der jede Aktivität i nach all ihren Vorgängeraktivitäten kommt, wird als Vorgänger-zulässig bezeichnet.

Tabelle 1 gibt ein einfaches Beispiel für die Transformation einer Lösung in eine Vorgänger-zulässige Lösung.

Tabelle 1: Beispiel der Konstruktion von Vorgänger-zulässigen Lösungen

Aktivitäten:	$J = 3$, daher $j = 1, 2, 3, 4, 5$	
Vorgängerbeziehungen:	$H = \{(1,2), (1,3), (1,4), (1,5), (2,4), (2,5), (3,5), (4,5)\}$ Anmerkung: $(2,4)$ ist dabei die einzige, nicht-triviale (d.h. Dummy-Aktivitäten betreffende) Beziehung	
Ursprungslösung:	$P = (3,1,5,4,2)$	$M = (1,3,2,1,1)$
Vorgänger-zulässige, äquivalente Lösung:	$P = (1,3,2,4,5)$	$M = (1,3,2,1,1)$

Um die äquivalente Lösung zu erhalten, wird schrittweise beginnend bei der Position mit der niedrigsten Priorität (d.h. ganz rechts) jede Aktivität so weit nach rechts gerückt, bis sich alle -Vorgängeraktivitäten dieser Aktivität in der Liste an früherer Stelle befinden, die Aktivität jedoch so nahe wie möglich an der ursprünglichen Position ist.

Prinzipiell ist die Forderung, dass nur Vorgänger-zulässige Lösungen betrachtet werden, nicht unbedingt nötig, da sich ohnehin die gleichen *makespans* ergeben. Dieses Vorgehen kann jedoch dazu benutzt werden, um Rechenzeit in den Algorithmen zu sparen, indem immer nur die Vorgänger-zulässigen Lösungen betrachtet werden. Allerdings muss dann jede Lösung zuerst in eine Vorgänger-zulässige umgewandelt werden, was wiederum etwas zusätzlichen Aufwand erzeugt. Im später beschriebenen genetischen Algorithmus wird auf die Konstruktion von Vorgänger-zulässigen Lösungen verzichtet, um die zufälligen Änderungen der Lösungen einfacher handhaben zu können.

Im VNS-Algorithmus (siehe später) wird das Vorgehen jedoch verwendet, um keine unnötigen Lösungen zu evaluieren.

1.2 Modellierung der Verteilungen der Bearbeitungszeiten

Wie oben bereits angemerkt, werden die Bearbeitungszeiten als zufällige Größen aus einem parametrischen Modell $d_{jm} \sim F(\theta_{jm})$ betrachtet. Dabei steht der Index j für die Aktivität und der Index m für den Modus, in dem die Aktivität ausgeführt wird. Es ist ersichtlich, dass eine Wahl für die Verteilungsfamilie F sowie eine Konstruktion des Parametervektors θ nötig ist. Die Verteilungsfamilie wird dabei für alle Aktivitäten als gleich angenommen. Der Parametervektor kann zwischen den Aktivitäten, sowie innerhalb einer Aktivität zwischen den Modi unterschiedlich sein. In dieser Arbeit werden als Verteilungen eine stetige Gleichverteilung, eine Dreiecksverteilung und eine (verschobene) Weibull-Verteilung untersucht. Die praktischen Überlegungen zu diesen Verteilungen folgen später in diesem Abschnitt.

1.2.1 Grundlegende Überlegungen

Um die jeweiligen Parameter zu bestimmen, wird folgende Überlegung angestellt: Es soll unter Angabe eines einzigen Input-Wertes B die Verteilung modelliert werden können. B kann als Basiszeit (also als Richtwert) verstanden werden. Dies folgt einer natürlichen Erweiterung der deterministischen Modellierung mit einer fixen Dauer für die Aktivitäten. Eine fixe Dauer ist in der Praxis sicher oft unrealistisch. Die zufällige Modellierung ist daher für die Umlegung solcher Probleme in die Praxis durchaus angebracht. Die Annahme, dass für die Bearbeitungsdauer einer Aktivität ein Richtwert angegeben werden kann, ist jedoch auch praktisch leicht denkbar. Dieser Richtwert entspricht nun der Basiszeit B und wird wie folgt in die Wahl der Parameter einbezogen.

Einleitend folgt eine Übersicht einiger Überlegungen, die für die Modellierung relevant sein werden:

- Es soll angenommen werden, dass für jede Aktivität eine erwartete Zeitdauer – eine Basiszeit B – angegeben werden kann. Dies entspricht der Dauer, die die Ausführung der Aktivität im Normalfall beansprucht.
- Diese erwartete Zeitdauer B kann über- oder unterschritten werden. Durch besondere Umstände kann es also zu einer beschleunigten oder verlängerten Bearbeitung der Aktivität kommen.

- Für jede Aktivität ist die Annahme einer unteren Grenze der Bearbeitungsdauer plausibel. Es wird also angenommen, dass es auch unter günstigsten Umständen nicht möglich ist, dass die Aktivität schneller als in dieser Mindestzeit erledigt wird. Diese Untergrenze hängt nun ebenfalls von der erwarteten Bearbeitungsdauer ab. Der Einfachheit halber wird angenommen, dass die Normbearbeitungszeit um höchstens 20 % unterschritten werden kann, d.h. die Mindestbearbeitungszeit bei $0.8 \cdot B$ liegt
- Die Streuung der Bearbeitungsdauer einer Aktivität hängt vorerst ebenfalls von der erwarteten Dauer ab. Für Jobs, die schnell erledigt sind, ist die Streuung der Dauern also geringer als für Jobs, die im Normalfall mehr Zeit beanspruchen.

Im Folgenden sei die Zufallsvariable der Bearbeitungszeit mit X bezeichnet. Es wird angenommen, dass diese durch eine differenzierbare Verteilungsfunktion $F(x)$ und somit über eine Dichte $f(x) = F'(x)$ beschrieben werden kann. Weiters sei die Basiszeit durch $B \in \mathbb{R}, 0 < B$ gekennzeichnet. Die Modellierungsvarianten folgen unter Berücksichtigung der oben angegebenen Punkte den folgenden Prinzipien:

(1a) Der Erwartungswert der Zufallsvariable entspricht (annähernd) der Basiszeit B.

$$\mathbb{E}(X) \approx B$$

(1b) Der plausibelste Wert (Modalwert) der resultierenden Zufallsvariable entspricht (annähernd) der Basiszeit B.

$$Mod(X) \approx B$$

(2) Die Dichte der Zufallsvariablen ist gleich null für Werte unter der Mindestzeit. Diese Mindestzeit ergibt sich dabei durch eine Beschleunigung um 20 % im Vergleich zur Basiszeit.

$$f(x) = 0, \forall x < 0.8 \cdot B$$

(3) Die Streuung der resultierenden Zufallszeiten hängt insofern von der Basiszeit B ab, als für Aktivitäten mit tendenziell längeren Bearbeitungsdauern eine größere Streuung der Zufallsvariablen angenommen wird.

$Var(X) = g(B)$ mit $g(\cdot): \mathbb{R}_+ \rightarrow \mathbb{R}_+$, monoton wachsend

Die formale Umsetzung sowie die Einbeziehung dieser Prinzipien wird im Folgenden für die drei berücksichtigen Verteilungen beschrieben.

1.2.2 Gleichverteilung

Es wird angenommen, dass es sich bei X um eine auf dem Intervall $[a, b]$ gleichverteilte Zufallsvariable handelt. a und b sollen dabei reelle Zahlen sein, sodass die Prinzipien (1)-(3) erfüllt sind. Allgemein lässt sich dann die Zufallsvariable X nach Forbes, Evans, Hastings, und Peacock (2010) folgendermaßen beschreiben:

$$X \sim U(a, b), \quad a, b \in \mathbb{R}, \quad a < b$$

$$f(x) = \begin{cases} \frac{1}{(b-a)} & , \quad a \leq x \leq b \\ 0 & , \quad sonst \end{cases}$$

$$\mathbb{E}(X) = \frac{a+b}{2}$$

$$Mod(X) = m, \forall\, m \in [a, b]$$

$$Var(X) = \frac{(b-a)^2}{12}$$

In diese Formulierung soll nun die Basiszeit B einfließen, wobei die Prinzipien (1)-(3) von oben berücksichtigt werden sollen. Es liegt nahe, folgende Wahl der Parameter a und b zu verwenden:

$$a = 0.8 \cdot B$$

$$b = 1.2 \cdot B$$

Somit gelten für diesen Fall, wo beide Parameter a und b nur von einer Größe B abhängen, folgende Eigenschaften der Verteilung:

$$X \sim U(a = 0.8 \cdot B, b = 1.2 \cdot B), \quad B \in \mathbb{R}_+$$

$$f(x) = \begin{cases} \frac{1}{(1.2 \cdot B - 0.8 \cdot B)} = \frac{1}{(0.4 \cdot B)} & , \quad 0.8 \cdot B \leq x \leq 1.2 \cdot B \\ 0 & , \quad sonst \end{cases}$$

$$\mathbb{E}(X) = \frac{0.8 \cdot B + 1.2 \cdot B}{2} = B$$

$$Mod(X) = m, \forall\, m \in [0.8 \cdot B, 1.2 \cdot B] \Rightarrow Mod(X) = B$$

$$Var(X) = \frac{(1.2 \cdot B - 0.8 \cdot B)^2}{12} = \frac{(0.4 \cdot B)^2}{12} = \frac{0.4^2 \cdot B^2}{12} = \frac{B^2}{75}$$

Damit sind die Bedingungen (1)-(3) erfüllt. Berechnet man den Variationskoeffizienten $c_v(X) = \frac{\sqrt{Var(X)}}{E(X)}$ ergibt sich für die Gleichverteilung der konstante Wert

$$c_v(X) = \frac{1}{\sqrt{75}} \approx 0.1155,$$

egal welches B eingesetzt wird.

Abbildung 1 zeigt die Dichtefunktion der Gleichverteilung für verschiedene Werte für B.

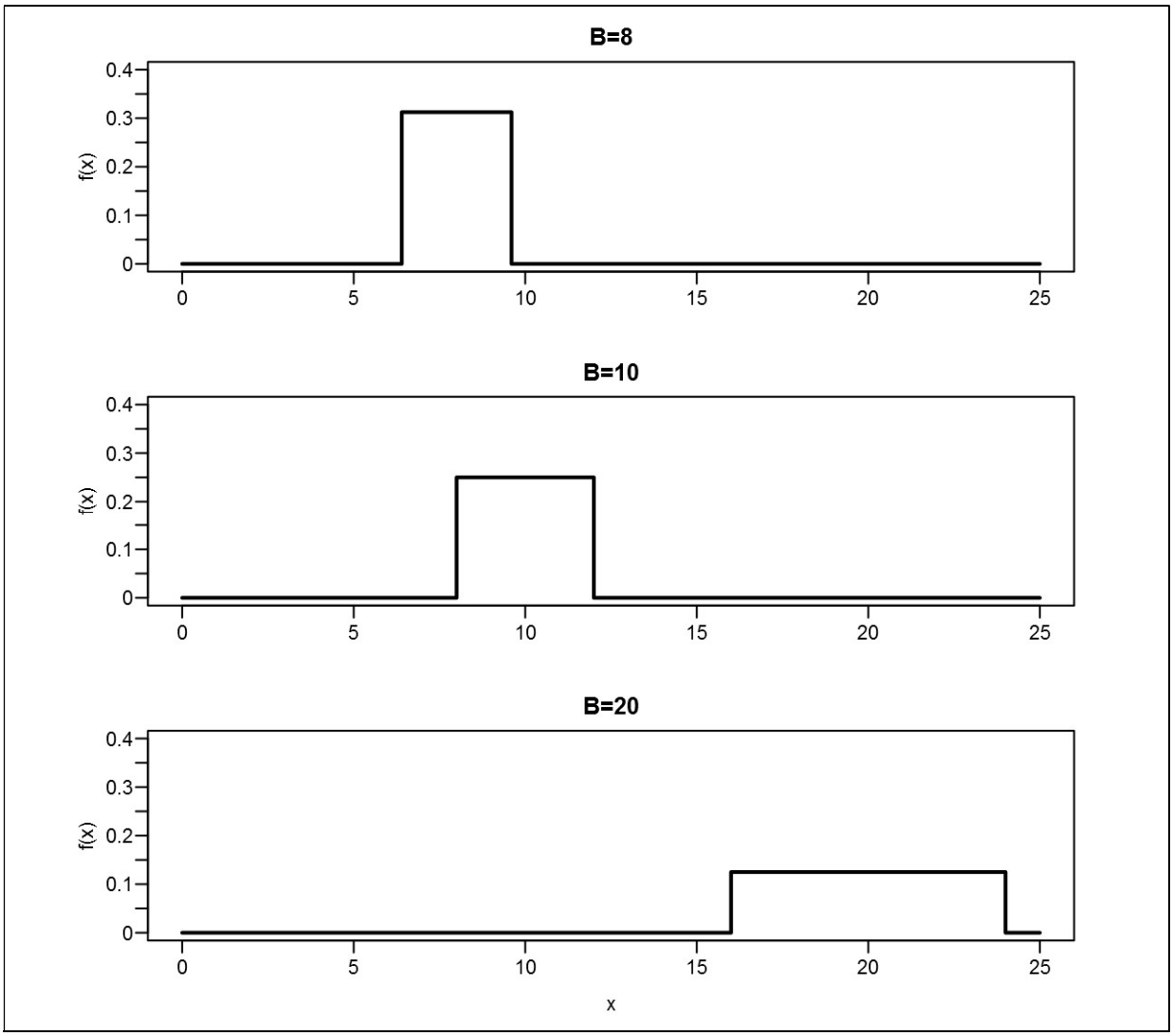

Abbildung 1: Dichtefunktion der Gleichverteilung für unterschiedliche Basiszeiten B.

Abbildung 1 veranschaulicht auch, welche Konsequenzen diese Verteilungsannahme hat. Die Basiszeit kann um 20 % über- oder unterschritten werden. Die Dichte verteilt sich dabei gleichmäßig über diesen Bereich. Dies zeigt jedoch auch die Schwächen dieser Verteilungsannahme. Einerseits macht die Annahme einer gleichmäßigen Verteilung der Auftrittswahrscheinlichkeiten über den gesamten Bereich möglicherweise nicht immer Sinn.

Vielmehr kann es sein, dass die Bearbeitungszeiten mit höherer Wahrscheinlichkeit in der Nähe des Erwartungswertes liegen und nur mit geringerer Wahrscheinlichkeit weiter davon entfernt sind. Eine gleichermaßen sinnvolle Annahme ist auch, dass die Basiszeit weiter überschritten als unterschritten werden kann. Möglicherweise ist eine Unterschreitung um mehr als 20 % nicht plausibel, eine Überschreitung des erwarteten Wertes um mehr als 20 % aber durchaus möglich. Dieses Problem ließe sich im Prinzip durch eine Gleichverteilung lösen, die beispielsweise zwischen 0.8·B und 1.4·B liegt. Allerdings verschiebt sich dann auch der Erwartungswert, sodass das Prinzip (1a) verletzt wird. Es folgt eine Modellierung, die unter Einhaltung der Prinzipien diesen Überlegungen gerecht wird.

1.2.3 Dreiecksverteilung

Es wird angenommen, dass die Zufallsvariable X eine allgemeine Form der Dreiecksverteilung annimmt. Es müssen drei reelle Parameter a, b und c gewählt werden, mit den Bedingungen $a < b$ und $a \leq c \leq b$.

Die Dreiecksverteilung lässt sich damit folgendermaßen beschreiben (Forbes et al, 2010):

$$X \sim Dreiecksverteilt(a, b, c), \quad a, b, c \in \mathbb{R}, \quad a < b \, , c \in [a, b]$$

$$f(x) = \begin{cases} \dfrac{2(x-a)}{(b-a)(c-a)} & , \quad a \leq x < c \\ \dfrac{2}{(b-a)} & , \quad x = c \\ \dfrac{2(b-x)}{(b-a)(b-c)} & , \quad c < x \leq b \\ 0 & , \quad sonst \end{cases}$$

$$\mathbb{E}(X) = \frac{a+b+c}{3}$$

$$Mod(X) = c$$

$$Var(X) = \frac{a^2 + b^2 + c^2 - ab - ac - bc}{18}$$

Die Kenntnis der Basiszeit B reicht hier noch nicht aus, um eine plausible Modellierung vorzunehmen. Es wird wieder angenommen, dass eine Unterschreitung der Basiszeit um 20 % möglich ist. Unter Berücksichtigung des oben angesprochenen Umstands, dass eine Überschreitung in höherem Ausmaß als eine Unterschreitung stattfinden soll, wird angenommen, dass die Bearbeitungszeit nach oben um mehr als 20 % abweichen kann. Eine

Abweichung nach oben um mehr als 40 % von der Basiszeit B wird als maximal mögliche Abweichung angenommen.

Somit ergibt sich eine plausible Modellierung dadurch, dass die Zufallsvariable dreiecksverteilt ist mit den Parametern $a = 0.8 \cdot B$, $b = 1.4 \cdot B$ und einem sinnvoll gewählten c. Zur Wahl des Parameters c sollen nun weitere Überlegungen angestellt werden. Setzt man wie im Prinzip (1a) angegeben voraus, dass $\mathbb{E}(X) = B$ sein soll, ergibt sich mit obigem a und b, dass $c = 0.8 \cdot B$. Somit gilt in dem Fall $a = c$, sodass ein Spezialfall der Dreiecksverteilung auftritt (siehe Abbildung 2).

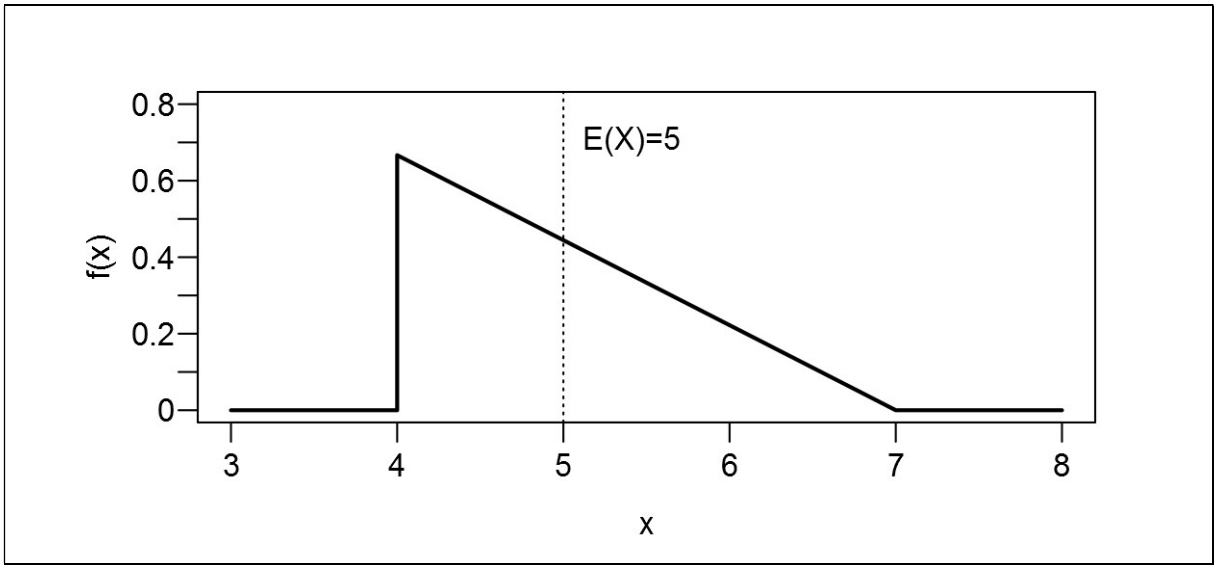

Abbildung 2: Dichtefunktion der Dreiecksverteilung falls $\mathbb{E}(X) = B$ und somit $a = c = 0.8 \cdot B$ am Beispiel $B = 5$.

Dann stimmt allerdings der Modalwert nicht mehr mit der Basiszeit B überein, wodurch das Prinzip (1b) verletzt ist. Für diese Modellierung ist der Modalwert $Mod(X) = c = 0.8 \cdot B$ (im Beispiel aus Abbildung 2 gilt $Mod(X) = 4$). Es scheint daher sinnvoll, eine andere Wahl für den Parameter c auf Basis der Prinzipien (1)-(3) zu treffen. Der nächste Versuch besteht darin, den Parameter nach dem Prinzip (1b) zu wählen. Wir nehmen also an, dass $Mod(X) = B$ gilt. Wegen $Mod(X) = c$ wird somit $c = B$ gesetzt.

Es ergibt sich dann folgende Modellierung für die Dreiecksverteilung:

$$X \sim Dreiecksverteilt(a = 0.8 \cdot B, b = 1.4 \cdot B, c = B), \quad B \in \mathbb{R}_+$$

$$f(x) = \begin{cases} \dfrac{2(x - 0.8 \cdot B)}{(1.4 \cdot B - 0.8 \cdot B)(B - 0.8 \cdot B)} = \dfrac{2(x - 0.8 \cdot B)}{0.12 \cdot B^2} & , \quad 0.8 \cdot B \leq x \leq B \\ \dfrac{2(1.4 \cdot B - x)}{(1.4 \cdot B - 0.8 \cdot B)(1.4 \cdot B - B)} = \dfrac{2(1.4 \cdot B - x)}{0.24 \cdot B^2} & , \quad B < x \leq 1.4 \cdot B \\ 0 & , \quad sonst \end{cases}$$

$$\mathbb{E}(X) = \frac{0.8 \cdot B + 1.4 \cdot B + B}{3} = \frac{16}{15} \cdot B$$

$$Mod(X) = B$$

$$Var(X) = \frac{(0.8 \cdot B)^2 + (1.4 \cdot B)^2 + B^2 - (0.8 \cdot B)(1.4 \cdot B) - (0.8 \cdot B) \cdot B - B \cdot (1.4 \cdot B)}{18}$$

$$= \frac{0.28 \cdot B^2}{18}$$

Es können somit für diese Wahl der Parameter die Bedingungen (1)-(3) als erfüllt betrachtet werden, sofern eine Abweichung des Erwartungswerts von B um $\frac{1}{15} \cdot B$ noch als annähernd gleich betrachtet wird. Weiters ergibt sich für den Variationskoeffizienten mit diesen Parametern folgendes Ergebnis:

$$c_v(X) = \frac{\sqrt{\frac{0.28 \cdot B^2}{18}}}{\frac{16}{15} \cdot B} \approx 0.1169,$$

Dies ähnelt dem Wert, der oben verwendeten Gleichverteilung. Die folgende Abbildung 3 zeigt die Dichtefunktion für verschiedene Beispiele der Basiszeit B.

Es ist ersichtlich, dass sich durch die Modellierung als Dreiecksverteilung ein Vorteil gegenüber der Gleichverteilung ergibt, da nun Werte um die Basiszeit B eine höhere Dichte aufweisen als am Rand. Trotzdem ist diese Modellierung noch insofern eingeschränkt, dass die Bearbeitungszeiten prinzipiell nach oben und unten beschränkt sind. Die Beschränkung nach unten kann in der Praxis durchaus sinnvoll sein, da Aktivitäten auch unter den günstigsten Umständen eine bestimmte Mindestzeit andauern. Die Bearbeitung einer Aktivität kann jedoch unter Umständen theoretisch beliebig lange dauern, sodass eine nach oben unbeschränkte Verteilung ebenfalls wünschenswert ist. Für diesen Zweck wird eine Modellierung auf Basis einer Weibull-Verteilung konstruiert.

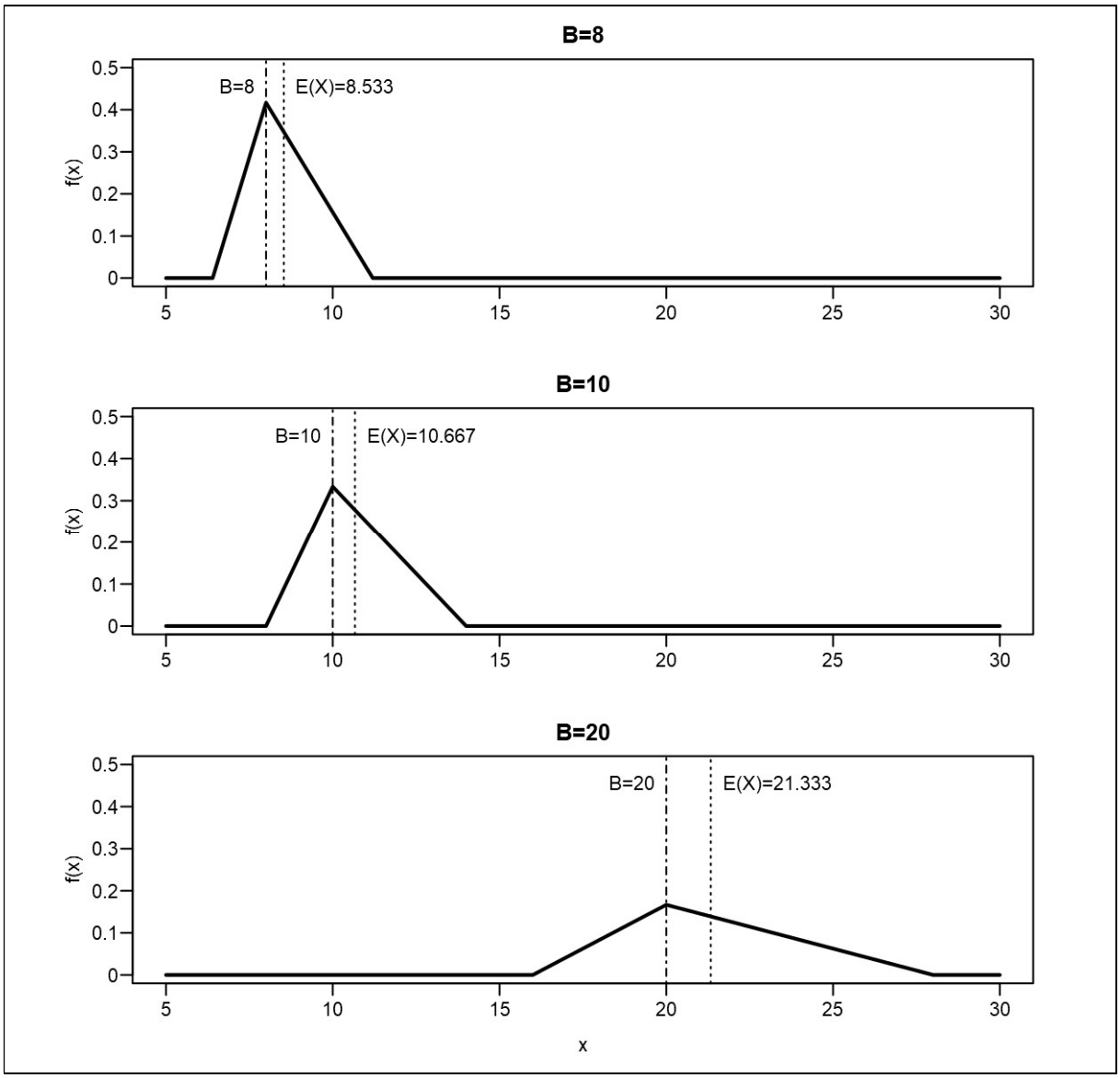

Abbildung 3: Dichtefunktion der Dreiecksverteilung für unterschiedliche Basiszeiten B.

1.2.4 Weibull-Verteilung

Die Weibull-Verteilung hat in der Standardversion die beiden reellen Parameter $\lambda > 0$ und $k > 0$ mit den positiven reellen Zahlen als Träger (Forbes et al, 2010). Die Weibull-Verteilung kann folgendermaßen beschrieben werden (Forbes et al, 2010):

$$X \sim Weibull(\lambda, k), \quad \lambda, k \in \mathbb{R}_+$$

$$f(x) = \begin{cases} \frac{k}{\lambda}\left(\frac{x}{\lambda}\right)^{k-1} \exp\left(-\left(\frac{x}{\lambda}\right)^{k}\right) & , \quad 0 \leq x \\ 0 & , \quad sonst \end{cases}$$

$$\mathbb{E}(X) = \lambda \cdot \Gamma\left(1 + \frac{1}{k}\right)$$

$$Mod(X) = \begin{cases} \lambda \cdot \left(\frac{k-1}{k}\right)^{\frac{1}{k}} & , \ 1 < k \\ 0 & , \ k \leq 1 \end{cases}$$

$$Var(X) = \lambda^2 \cdot \left(\Gamma\left(1+\frac{2}{k}\right) + \left[\Gamma\left(1+\frac{1}{k}\right)\right]^2\right)$$

mit der Gamma-Funktion

$$\Gamma(x) = \int_0^\infty t^{x-1} \exp(-t)\, dt$$

Unter Kenntnis der Basiszeit B, der Annahme (2), dass die Zufallsvariable jedenfalls erst Werte über $0.8 \cdot B$ mit positiver Wahrscheinlichkeit annimmt und der Annahme (1b), dass der Modus der Verteilung in etwa der Basiszeit B entspricht, wird die folgende Modellierung konstruiert. Wir betrachten die Zufallsvariable X, welche sich folgendermaßen ergibt:

$$X = 0.8 \cdot B + Y \text{ mit } Y \sim Weibull\left(\lambda = 0.2 \cdot B \cdot \left(\frac{k-1}{k}\right)^{-\frac{1}{k}}, k\right), \quad k > 1$$

Es handelt sich also um eine um den Wert $0.8 \cdot B$ nach oben verschobene Weibullverteilung. Alternativ kann diese Zufallsvariable über folgende Dichte in Anlehnung an die oben beschriebene Weibull-Verteilung beschreiben werden:

$$f(x) = \begin{cases} \frac{k}{\lambda}\left(\frac{x-0.8\cdot B}{\lambda}\right)^{k-1} \exp\left(-\left(\frac{x-0.8\cdot B}{\lambda}\right)^{k}\right) & , \quad 0.8 \cdot B \leq x \\ 0 & , \quad sonst \end{cases}$$

mit

$\lambda = 0.2 \cdot B \cdot \left(\frac{k-1}{k}\right)^{-\frac{1}{k}}$ und $k > 1$ beliebig.

Der Parameter λ ist dabei so gewählt, dass sich wie unten ersichtlich $Mod(X) = B$ ergibt.

Damit ist Bedingung (2) erfüllt. Außerdem ergeben sich folgende Eigenschaften:

$$\mathbb{E}(X) = B \cdot \left[0.8 + 0.2 \cdot \left(\frac{k-1}{k}\right)^{-\frac{1}{k}} \cdot \Gamma\left(1+\frac{1}{k}\right)\right]$$

$$Mod(X) = B$$

$$Var(X) = B^2 \cdot 0.04 \cdot \left[\left(\frac{k-1}{k}\right)^{-\frac{2}{k}} \cdot \left(\Gamma\left(1+\frac{2}{k}\right) + \left[\Gamma\left(1+\frac{1}{k}\right)\right]^2\right)\right]$$

Damit sind die Bedingungen (1b), und (3) erfüllt. Bedingung (1a) ist erfüllt, wenn gilt

$$\left[0.8 + 0.2 \cdot \left(\frac{k-1}{k}\right)^{-\frac{1}{k}} \cdot \Gamma\left(1 + \frac{1}{k}\right)\right] \approx 1.$$

Abbildung 4, zeigt diesen Faktor, in Abhängigkeit vom Parameter k. Man sieht, dass für kleine Werte von k eine stärkere Abweichung des Erwartungswerts vom Wert B eintritt.

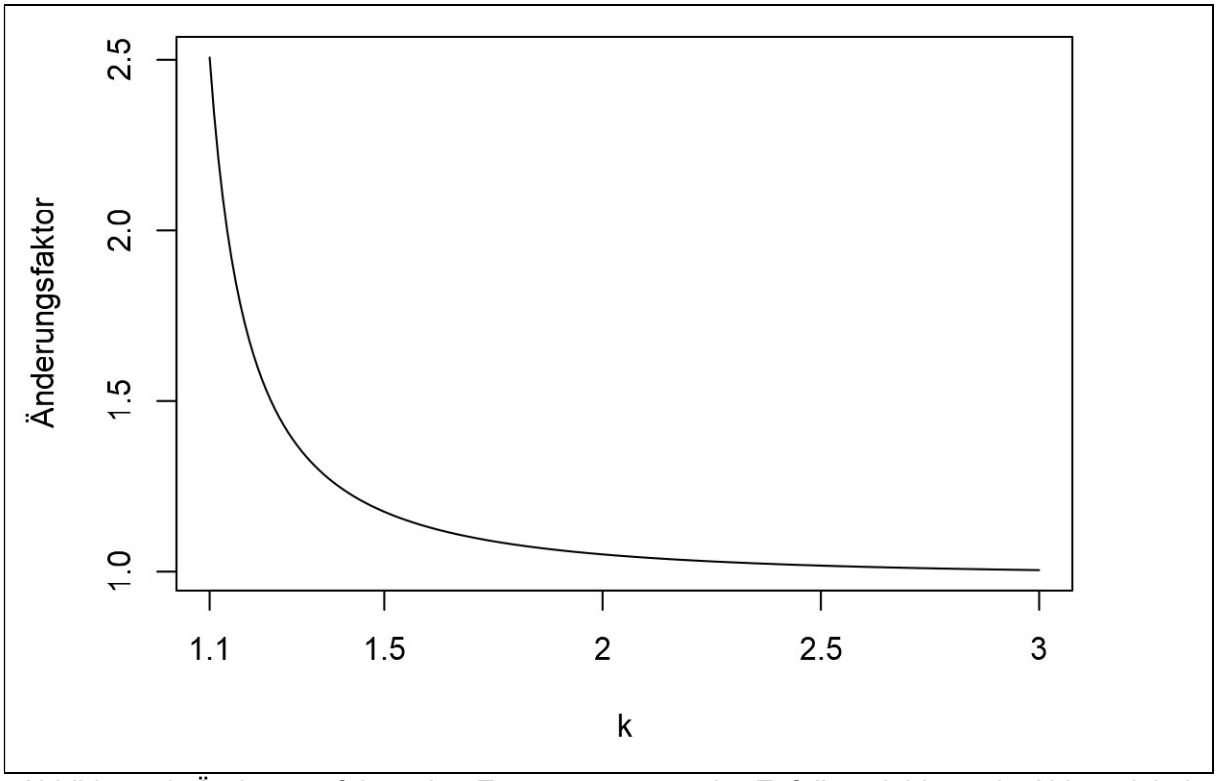

Abbildung 4: Änderungsfaktor des Erwartungswerts der Zufallsvariablen X in Abhängigkeit des Parameters k.

Zu beachten ist, dass sich die Form der Verteilung ebenfalls mit dem Parameter k ändert. Letztendlich wird ein Wert von $k = 1.8$ gewählt. Dieser Wert erhält die angestrebte Form der Verteilung und führt zu einer tragbaren Abweichung des Erwartungswertes der Verteilung von der Basiszeit B. Die Abweichung beträgt für diesen Fall ca. $\frac{1}{13} \cdot B$. Die resultierende Verteilung ist für einige beispielhafte Werte für B in Abbildung 5 dargestellt.

Zusammenfassend wird also für die dritte Art der Modellierung folgende Zufallsvariable X herangezogen:

$$X = 0.8 \cdot B + Y$$

mit

$$Y \sim Weibull\left(\lambda = 0.2 \cdot B \cdot \left(\frac{1.8-1}{1.8}\right)^{-\frac{1}{1.8}}, k = 1.8\right), B > 0$$

bzw.

$$Y \sim Weibull(\lambda \approx 0.3138 \cdot B, k = 1.8), B > 0$$

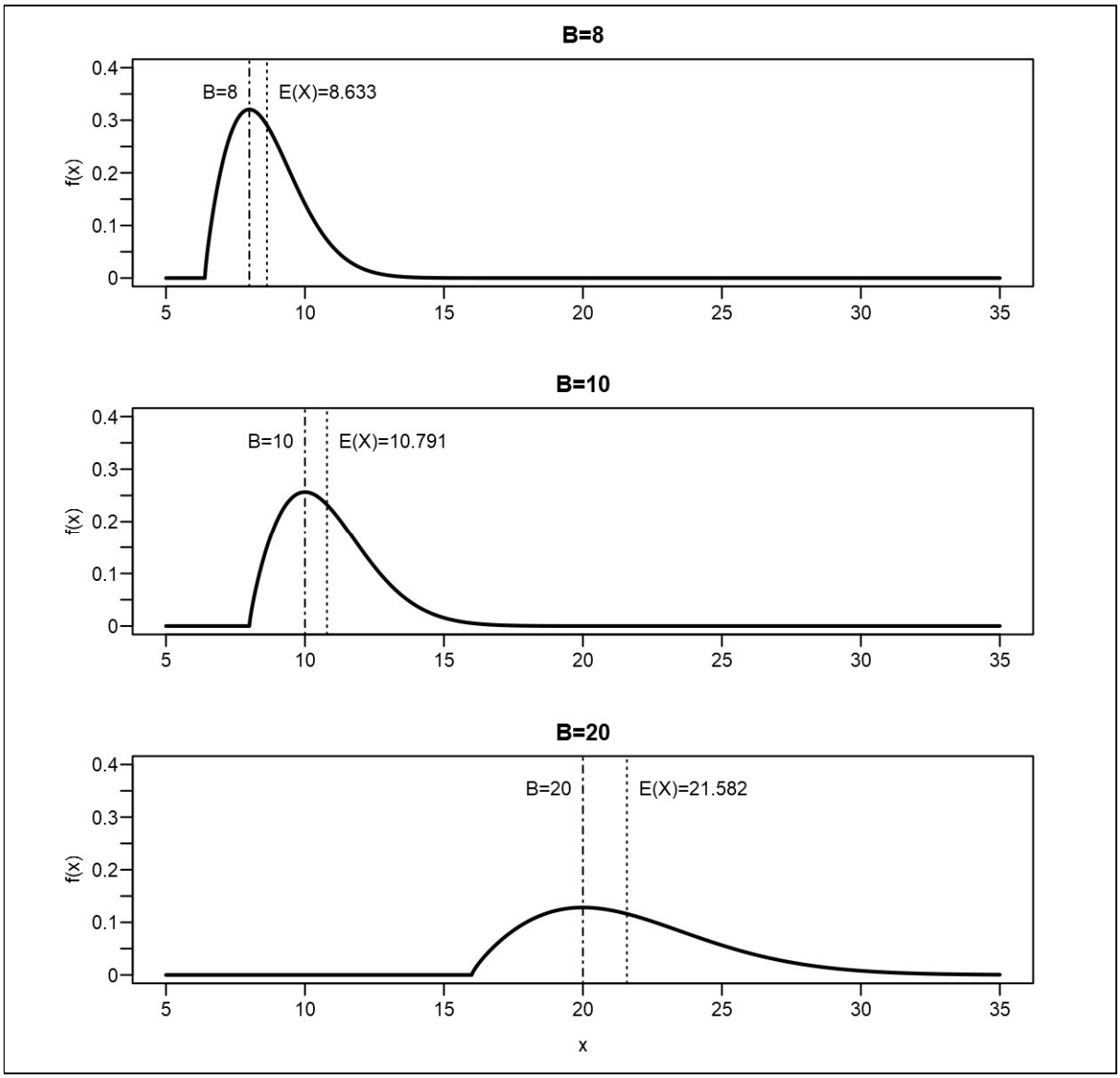

Abbildung 5: Dichtefunktion auf Basis der Weibull-Verteilung für unterschiedliche Basiszeiten B.

Die auf diese Weise konstruierten Zufallsvariablen werden zur Simulation der Szenarien verwendet. In jedem Fall reicht die Kenntnis einer Basiszeit aus, um die Modellierung und damit die Simulation durchführen zu können.

1.3 Datengrundlage

1.3.1 PSPLIB

Als Datengrundlage werden Probleminstanzen aus der PSPLIB (Kolisch & Sprecher, 1996) verwendet. Diese sind auf http://www.om-db.wi.tum.de/psplib/main.html frei verfügbar und wurden bereits in unterschiedlichen Arbeiten (z.B.: Kolisch & Drexl, 1997; Özdamar, 1999; Hartmann 2001; Alcaraz, Maroto & Ruiz, 2003) verwendet.

Verwendet werden in dieser Arbeit je 30 zufällig ausgewählte Instanzen aus den Kategorien J10, J16 und J30 der Multi-mode-Problemstellungen. Die Übersicht der analysierten Projekte findet sich in Tabelle 14 im Appendix (Abschnitt 6.2). Es werden lediglich Instanzen ausgewählt, in denen die Anforderungen an die erneuerbaren Ressourcen in allen Modi eingehalten werden können.

Tabelle 2 gibt einen Überblick über die in den Instanzen der PSPLIB enthalten Informationen. Es werden nur diejenigen Informationen angegeben, die in der vorliegenden Arbeit Verwendung finden.

Tabelle 2: Aus den Datensätzen der PSPLIB extrahierte Informationen

Information	Beschreibung
Vorgängerbeziehungen:	Unmittelbare Nachfolgeaktivitäten je Aktivität
Modi:	Anzahl an Modi je Aktivität ($M_j = 1$ für die Dummy-Aktivitäten und $M_j = 3$ für alle anderen Aktivitäten)
Bearbeitungsdauern:	ganzzahlige Bearbeitungsdauern pro Aktivität und Modus
Ressourcentypen:	$K = 2$ erneuerbare und $N = 2$ nicht erneuerbare Ressourcen
Ressourcenbedarf:	ganzzahliger Ressourcenverbrauch pro Ressource, Aktivität und Modus
Ressourcenkapazität:	ganzzahlige Ressourcenkapazität pro Ressource

Die Instanzen werden insofern erweitert, indem für jedes Projekt eine Sammlung von $N_{scen} = 100$ Szenarien für alle drei in Abschnitt 1.2 beschriebenen Zufallsmodelle konstruiert werden. Dabei werden die aus der PSPLIB extrahierten Bearbeitungsdauern als Basiszeiten B für die Simulation der Zufallsvariablen herangezogen. Dabei ist zu beachten, dass die Zufallsvariablen so konstruiert sind, dass die Angabe einer Basiszeit B ausreicht, um eine Realisierung der Zufallsvariable zu simulieren.

Beispielsweise beträgt die aus der PSPLIB für das Projekt j1010_7 entnommene Bearbeitungsdauer von Aktivität $j = 2$ im dritten Modus $d_{23} = 10$ Einheiten. Für jedes Szenario wird diese Bearbeitungszeit als Basiszeit $B = 10$ gesetzt und die Bearbeitungsdauer d_{23} als eine Zufallsvariable X aus den folgenden Verteilungen generiert:

Gleichverteilung:	$X \sim U(a = 8, b = 12)$
Dreiecksverteilung:	$X \sim Dreiecksverteilt(a = 8, b = 14, c = 10)$
Weibull-Verteilung:	$X = 8 + Y$ mit $Y \sim Weibull(\lambda \approx 3.138, k = 1.8)$

Es ergeben sich insgesamt 270 (je 30 Projekte aus J10, J16 und J30 mit je drei Verteilungsmodellen) Probleminstanzen, mit je 100 Szenarien.

1.3.2 Anzahl an möglichen Lösungen

Ohne Berücksichtigung der Vorgängerbeziehungen beträgt die Anzahl an möglichen Lösungen für die Problemstellungen dieser Art

$$\#\{\text{Lösungen}\} = (J + 2)! \cdot \prod_{j=0}^{J+1} M_j$$

Betrachtet man die Anzahl an möglichen Lösungen für die hier betrachteten Probleme, so ergibt sich die in Tabelle 3 angegebenen Anzahl an möglichen Lösungen

Tabelle 3: Anzahl an möglichen Lösungen (ohne Vorgängerbeziehungen) für die behandelten Projekt-Größen

Anzahl an Aktivitäten	Anzahl an möglichen Lösungen
$J = 10$	$\approx 2.82846 \cdot 10^{13}$
$J = 16$	$\approx 2.75601 \cdot 10^{23}$
$J = 30$	$\approx 5.41763 \cdot 10^{49}$

2 Methode

Für die Optimierung werden zwei heuristische Lösungsalgorithmen herangezogen. Heuristische Algorithmen führen nicht zwangsweise zum Auffinden einer optimalen Lösung, stellen für viele Problemstellungen jedoch die einzige Alternative dar. Vor allem können heuristische Algorithmen praktisch immer eingesetzt werden, wenn die Anzahl an möglichen Lösungen zu groß oder aufgrund der Problemstellung keine realisierbaren exakten Optimierungsverfahren angewandt werden können.

Als heuristische Algorithmen werden zwei häufig benutzte Methoden verwendet: eine *Variable Neighbourhood Search* (VNS) und ein genetischer Algorithmus (GA). Die Umsetzung dieser beiden Algorithmen wird im folgenden Abschnitt beschrieben.

2.1 VNS-Algorithmus

2.1.1 Grundidee

Die Grundidee der VNS ist nach Hansen, Mladenović, und Moreno Pérez (2010), mit einer Startlösung zu beginnen, und in deren Umgebung nach einer Nachbarlösung zu suchen, die den Zielfunktionswert verbessert (lokale Suche). Eine Nachbarlösung ist dabei eine Lösung, die durch einige wenige Transformationsschritte aus der Ursprungslösung konstruiert werden kann. Wird eine bessere Lösung gefunden, wird diese als neue beste Lösung aktualisiert. Nun wird wiederum in der Nachbarschaft dieser Lösung nach einer besseren Lösung gesucht, usw. Ist eine Lösung die beste in ihrer Nachbarschaft, werden schrittweise Bereiche von immer größer werdenden Umgebungen abgesucht, bis wieder eine bessere Lösung gefunden wird (Erweiterung der Nachbarschaft). Anschließend wird auf diese Lösung wieder die lokale Suche angewandt, usw.

Dieses Vorgehen beinhaltet nach Hansen et al (2010) zwei grundlegende Prinzipien, die Anwendung finden. Die jeweils aktuell beste Lösung ist ein lokales Optimum (in Bezug auf die Nachbarschaft), da sie eben als beste Lösung innerhalb ihrer Nachbarschaft gewählt wird. Die Erweiterung der Nachbarschaft führt dazu, dass ein lokales Optimum auch wieder verlassen werden kann. Die VNS ist sehr flexibel einsetzbar und hat sich bei vielen Optimierungsproblemen bewährt (Hansen et al, 2010).

2.1.2 Die Nachbarschaften

Die Verwendung der VNS benötigt in erster Linie eine Definition von Nachbarschaften. Diese sollen unterschiedliche Größen (im Sinne der Menge enthaltener Lösungen) aufweisen. Für die hier behandelten Problemstellungen werden die im Folgenden beschriebenen zwei Arten von Nachbarschaften verwendet.

2.1.2.1 Unmittelbare Nachbarschaft einer Lösung

Da die Berechnung der Zielfunktion mit einem hohen Aufwand verbunden ist, wird für die lokale Suche eine relativ enge Umgebung um die Lösung konstruiert. Da eine Lösung jeweils aus einer Prioritätsliste P und einer Modusliste M besteht, müssen beide Teile in die Konstruktion der Nachbarschaft einbezogen werden. Für die Konstruktion der unmittelbaren Nachbarschaft wird ein mehrstufiges Vorgehen gewählt. y bezeichnet in den folgenden Ausführungen eine Lösung, also ein Paar aus P und M.

Schritt 1: Konstruktion einer (unmittelbaren) Nachbarschaft $N_P(y)$ unter Berücksichtigung der Prioritätsliste P.

Schritt 2: Konstruktion einer (unmittelbaren) Nachbarschaft $N_M(y')$ unter Berücksichtigung der Modusliste M für jedes $y' \in N_P(y)$.

Schritt 3: Vereinigung aller in Schritt 2 erstellten Nachbarschaften zur unmittelbaren Nachbarschaft $N_0(y)$.

Ad Schritt 1:

Für die Konstruktion von $N_P(y)$ wird die Modusliste M der Lösung y vorerst unverändert gelassen. Die Prioritätsliste P wird so abgeändert, dass genau eine Aktivität herausgeschnitten, und an eine beliebige Stelle positioniert wird. Eine solche Transformation wird häufig als *insertion move* bezeichnet (z. B. Liaw, 2000). Tabelle 4 gibt ein Beispiel für die Konstruktion einer solchen Umgebung.

Tabelle 4: Beispiel der Konstruktion von $N_P(y)$, der unmittelbaren Nachbarlösungen in Bezug auf die Prioritätsliste einer Lösung y

Aktivitäten:	$J = 3$, daher $j = 1, 2, 3, 4, 5$ (1 und 5 sind Dummy-Aktivitäten)
Vorgängerbeziehungen:	$H = \{(1,2), (1,3), (1,4), (1,5), (2,5), (3,5), (4,5)\}$ Anmerkung: nur triviale Vorgängerbeziehungen
Ursprungslösung:	$P = (1, 3, 2, 4, 5)$ $M = (1, 1, 2, 1, 1)$
Nachbarlösungen: • nur Prioritätslisten • Unterstrichene Indizes markieren die neu positionierten Aktivitäten • Mit * markierte Lösungen sind nicht redundant (Vorgängerbeziehungen)	*$(1, 3, 2, 4, 5)$; $(3, \underline{1}, 2, 4, 5)$; $(3, 2, \underline{1}, 4, 5)$; $(3, 2, 4, \underline{1}, 5)$; $(3, 2, 4, 5, \underline{1})$; *$(1, 2, \underline{3}, 4, 5)$; *$(1, 2, 4, \underline{3}, 5)$; $(1, 2, 4, 5, \underline{3})$; $(\underline{2}, 1, 3, 4, 5)$; *$(1, 3, 4, \underline{2}, 5)$; $(1, 3, 4, 5, \underline{2})$; $(\underline{4}, 1, 3, 2, 5)$; *$(1, \underline{4}, 3, 2, 5)$; $(1, 3, 2, 5, \underline{4})$; $(\underline{5}, 1, 3, 2, 4)$; $(1, \underline{5}, 3, 2, 4)$; $(1, 3, \underline{5}, 2, 4)$

Zu beachten ist einerseits, dass die Aktivitäten auch an die ursprüngliche Stelle rücken können. Daher ist die Ursprungslösung selbst immer in dieser Nachbarschaft enthalten. Wegen den bestehenden Vorgängerbeziehungen zwischen den Aktivitäten können aufgrund der Vorgänger-Zulässigkeit bestimmte Lösungen der Nachbarschaft als redundant betrachtet werden. Beispielsweise können auf jeden Fall die Dummy-Aktivitäten immer an die erste und letzte Position rücken. Die so erstellten Nachbarlösungen werden daher noch in Bezug auf die Vorgänger-Zulässigkeit verändert. In Tabelle 4 sind die Lösungen, die nach dieser Reduktion in der Nachbarschaft verbleiben, mit einem * gekennzeichnet. Dabei wird angenommen, dass nur die trivialen Vorgänger-Beziehungen mit den Dummy-Aktivitäten 1 und 5 bestehen. Gibt es außer bei den Dummy-Aktivitäten keinerlei Vorgängerbeziehungen, und ist die Ursprungslösung bereits Vorgänger-zulässig, dann beträgt die Menge an Lösungen in dieser Nachbarschaft

$$\#N_P(y) = 1 + J \cdot (J-1) - (J-1) = 1 + (J-1)^2$$

Die Menge an Nachbarlösungen reduziert sich durch das Vorgehen also beträchtlich. In den realen Problemen würden sich die Anzahl aufgrund zusätzlicher, nicht-trivialer Vorgängerbeziehungen noch weitere Lösungen entfernen lassen. Auf jede der nach der Reduktion verbleibenden Nachbarlösungen $y' \in N_P(y)$ wird dann Schritt 2 angewandt.

Ad Schritt 2:

Für die Konstruktion von $N_M(y')$ wird die Prioritätsliste von y' unverändert gelassen. In der Liste der Modi wird der Modus genau einer Aktivität durch einen zulässigen Modus der

entsprechenden Aktivität ersetzt. Somit beinhaltet diese Nachbarschaft einer Lösung all jene Lösungen mit gleicher Prioritätsliste, in denen der Modus höchstens einer Aktivität anders gewählt ist. Dieser Schritt wird im binären Fall (zwei Modi) häufig als *bit flip* bezeichnet (z.B.: Sivanandam & Deepa, 2008) und lässt sich leicht auf einen allgemeinen *flip move* mit mehreren Modi verallgemeinern. Da dieser Schritt in diesem Fall auf die Modi angewandt wird, wird im Folgenden dafür die Bezeichnung *mode flip* verwendet. Tabelle 5 gibt ein Beispiel für eine Lösung und deren Nachbarschaft für ein Problem mit $J = 3$ Aktivitäten und $M_j = 3$ Modi pro Nicht-Dummy-Aktivität. Zu beachten ist, dass es sich bei den Aktivitäten mit den Indizes 1 und 5 um die Dummy-Aktivitäten mit nur einem Modus handelt.

Tabelle 5: Beispiel der Konstruktion von $N_M(y')$, der unmittelbaren Nachbarlösungen in Bezug auf die Modusliste einer Lösung y'

Aktivitäten:	$J = 3$, daher $j = 1, 2, 3, 4, 5$ (1 und 5 sind Dummy-Aktivitäten)
Modi:	$M_1 = M_5 = 1$; $M_2 = M_3 = M_4 = 3$
Ursprungslösung:	$P = (1, 3, 2, 4, 5)$ $M = (1, 1, 2, 1, 1)$
Nachbarlösungen:	$(1, 1, 2, 1, 1)$;
• Nur Moduslisten	$(1, \underline{2}, 2, 1, 1)$; $(1, \underline{3}, 2, 1, 1)$; $(1, 1, \underline{1}, 1, 1)$;
• Unterstrichen ist der geänderte Modus	$(1, 1, \underline{3}, 1, 1)$; $(1, 1, 2, \underline{2}, 1)$; $(1, 1, 2, \underline{3}, 1)$

Auch die Ursprüngliche Lösung ist wieder Teil der Nachbarschaft. Dann gilt für die Anzahl der Lösungen in dieser Nachbarschaft, dass

$$\#N_M(y') = 1 + \sum_{j=0}^{J+1} (M_j - 1)$$

In den in dieser Arbeit verwendeten Problemstellungen ist $M_j = 3$ für $j = 1, \dots, J$ und $M_0 = M_{J+2} = 1$. Somit enthalten diese Nachbarschaften $1 + 2 \cdot 10 = 21$, $1 + 2 \cdot 16 = 33$ bzw. $1 + 2 \cdot 30 = 61$ Lösungen für die J10-, J16- bzw. J30-Probleminstanzen.

ad Schritt 3:

Die beiden Schritte werden nun kombiniert, sodass eine unmittelbare Nachbarschaft bezüglich beider Teile (Prioritätsliste und Modusliste) einer Lösung y entsteht. Diese Menge an Lösungen wird ab jetzt mit $N_0(y)$ bezeichnet und lässt sich mit den oben beschriebenen beiden Schritten folgendermaßen beschreiben

$$N_0(y) = \{N_M(y') \colon y' \in N_P(y), y' \text{ Vorgänger-zulässig}\}$$

2.1.2.2 Erweiterte Nachbarschaft einer Lösung

Der später beschriebene VNS-Algorithmus verwendet noch eine weitere Definition von Nachbarschaften. Ziel ist es dabei, nicht mehr nur die nächsten Lösungen, sondern potentiell auch weiter entfernte Lösungen in die Nachbarschaft miteinzubeziehen. Weiter entfernt meint, dass sich die Lösungen zwar noch durch einfache Transformationen aus der Ursprungslösung erzeugen lassen, allerdings mehr solche Transformationen angewandt werden. Die Weite der Nachbarschaft soll dabei von einem Parameter k abhängen, welcher eben die Distanz der Lösung zur Ursprungslösung steuert. Wiederum muss der zweiteilige Typ der Lösungen (Prioritätsliste und Modusliste) berücksichtigt werden. Wir betrachten nun vorerst erweiterte Nachbarschaften in Bezug auf diese beiden Teile getrennt. Diese werden dann zu einer gemeinsamen Repräsentation einer erweiterten Nachbarschaft kombiniert.

Erweiterte k-Nachbarschaft bezüglich Prioritätsliste

Wir bezeichnen diese Nachbarschaft einer Lösung y mit $N_P^{(k)}(y)$ und erzeugen diese, indem wir nacheinander k *insertion moves* (siehe oben) auf die Ursprungsprioritätsliste ausführen. Dabei wird eine Aktivität aus der Prioritätsliste gewählt. Diese wird aus der Liste herausgeschnitten und an einer anderen Position wieder eingefügt. Die Reihenfolge der verbleibenden Aktivitäten bleibt dabei erhalten. Es kann auch die Ursprungsposition gewählt werden. In der k-Nachbarschaft wird dieser Schritt k mal wiederholt.

Erweiterte k-Nachbarschaft bezüglich Modusliste

Wir bezeichnen diese Nachbarschaft einer Lösung y mit $N_M^{(k)}(y)$ und erzeugen diese, indem wir nacheinander k *mode flips* (siehe oben) ausführen. Dabei wird der Modus genau einer Aktivität durch einen zulässigen Modus dieser Aktivität ersetzt. Wiederum kann auch der Ursprungsmodus gewählt werden. In der k-Nachbarschaft wird dieser Schritt k mal wiederholt.

Kombination der beiden Vorgehensweisen

Die erweiterte k-Nachbarschaft $N^{(k)}(y)$ einer Lösung y in Bezug auf beide Teile der Lösung entsteht nun, indem insgesamt k Schritte bei beiden Teilen (Prioritätsliste und Modusliste) vorgenommen werden. Wir können dies also durch folgendes Vorgehen beschreiben:

$$N^{(k)}(y) = \left\{ y'' \in N_M^{(k'')}(y') : y' \in N_P^{(k')}(y), k' + k'' = k \right\}$$

Einige Anmerkungen zu den Nachbarschaften:

Es ist zu beachten, dass die Nachbarschaften mit steigendem k schnell größer werden. Im Prinzip enthält die Nachbarschaft $N^{(k)}(y)$ für $k \geq 2 \cdot J$ alle Lösungen des Lösungsraums. Die Nachbarschaften sind außerdem ineinander verschachtelt, d.h. für k ≥ 1: $N^{(k)}(y) \subseteq N^{(k+1)}(y)$.

Im Algorithmus wird ein Vorgehen gewählt, welches unter Vorgabe des Parameters k und der Ursprungslösung eine Lösung zufällig aus der entsprechenden Nachbarschaft zieht. Zu beachten ist allerdings, dass bei diesem Vorgehen nicht jede Lösung gleich wahrscheinlich ist, da die Schritte sequentiell durchgeführt werden. Das genaue Vorgehen ist in der später angegebenen Formulierung des Algorithmus in Pseudo-Code beschrieben.

2.1.3 Umsetzung des VNS Algorithmus

Die hier angegebene Umsetzung folgt im Prinzip dem in Hansen et al (2010) als *basic VNS* beschriebenen Algorithmus. Ein solcher Algorithmus besteht nach Hansen et al (2010) im Wesentlichen aus zwei Teilen:

1.) Sequentielle Anwendung einer lokalen Suche (*local search*) in einer Nachbarschaft um die aktuell beste Lösung bis ein lokales Optimum erreicht ist.
2.) Erweiterung der Suche auf eine erweiterte Nachbarschaft der aktuell besten Lösung (*shaking*), um das lokale Optimum gegebenenfalls wieder verlassen zu können.

Diesen beiden Schritten sind die beiden Algorithmen A7 und A9 zugeordnet. Diese werden im Folgenden beschrieben, wobei die grundlegende Struktur von Hansen et al (2010) auf die hier vorliegende Problemstruktur und die oben beschriebenen Konstruktionen der Nachbarschaften übertragen wird. Die Anwendung der lokalen Suche erfolgt dabei nach dem von Hansen et al (2010) als *best improvement* bezeichneten Prinzip. Dabei wird immer die gesamte definierte Nachbarschaft evaluiert, und die beste darin gefundene Lösung als neue beste Lösung gewählt.

Da der Algorithmus verschiedene Transformationen bzw. Teilschritte beinhaltet, erfolgt die Beschreibung hierarchisch. Es werden die einzelnen Teilschritte und Funktionen gesondert beschrieben und im eigentlichen VNS-Algorithmus (Algorithmus A10) wie angegeben verwendet.

Algorithmus A1: Aktualisierung Prioritätsliste – $replace\_P(y, P')$

Input: $y = (P, M)$ … Ursprungslösung

P' … Neue Prioritätsliste

1. $y' \leftarrow (P', M)$

2. return y'

Der Algorithmus A1 tauscht die Prioritätsliste P einer Lösung y mit einer neuen Prioritätsliste P' aus. Gleiches gilt für die Moduslisten in Algorithmus A2.

Algorithmus A2: Aktualisierung Modusliste – $replace\_M(y, M')$

Input: $y = (P, M)$ … Ursprungslösung

M' … Neue Modusliste

1. $y' \leftarrow (P, M')$

2. return y'

Der in Abschnitt 2.1.2.1 beschriebene *insertion move*, welcher auf die Ursprungslösung y angewandt wird, ist in Algorithmus A3 beschrieben. Die Auswahl der auszutauschenden Positionen erfolgt dabei zufallsbasiert.

Algorithmus A3: *insertion move* – $insertion\_move(y)$

Input: $y = (P, M)$ … Ursprungslösung mit $P = (p_1, p_2, \dots, p_J)$

1. $J \leftarrow$ Anzahl Aktivitäten in y

2. Wähle $i \in \{1, 2, \dots, J\}$ zufällig

3. Entferne Aktivität p_i an Position i aus P

$$P' = (p_1, p_2, p_{i-1}, p_{i+1}, \dots, p_J) = (p'_1, p'_2, \dots, p'_{J-1})$$

4. Wähle $i' \in \{1, 2, \dots, J\}$ zufällig

5. Setze die entfernte Aktivität in P' an Stelle i' wieder ein

$$P'' = ({p'}_1, {p'}_2, {p'}_{i'-1}, {p'}_{i'}, \dots, {p'}_{J-1}) = ({p''}_1, {p''}_2, \dots, {p''}_J)$$

6. Ersetze Prioritätsliste in y durch P''

$$y' \leftarrow replace\_P(y, P'')$$

7. return y'

Die zweite für die Erstellung der Nachbarlösungen notwendige Transformation ist der in Abschnitt 2.1.2.1 beschriebene *mode flip*. Dieser ist in Algorithmus A4 beschrieben. Dabei erfolgen die Auswahl der Aktivität, deren Modus geändert wird, und die Wahl des neuen Modus wieder zufallsbasiert.

Algorithmus A4: mode-flip-move – $mode\_flip\_move(y, M_{possible})$

Input: $y = (P, M)$ … Ursprungslösung mit $M = (m_1, m_2, \dots, m_J)$

$M_{possible} = \{M_1, \dots, M_J\}$ …Mögliche Ausführungsmodi der Aktivitäten

1. $J \leftarrow$ Anzahl Aktivitäten in x

2. Wähle $i \in \{1, 2, \dots, J\}$ zufällig

3. Wähle ${m'}_i \in 1, \dots, M_i$ zufällig

4. Ersetze Modus m_i durch ${m'}_i$

$$M' = (m_1, \dots, m_{i-1}, {m'}_i, m_{i+1}, \dots, m_J)$$

5. Ersetze Modusliste in y durch M'

$$y' \leftarrow replace\_M(y, M')$$

6. return y'

Diese grundlegenden Transformationen der Lösungen können nun verwendet werden, um die in Abschnitt 2.1.2 beschriebenen Nachbarschaften zu konstruieren. Algorithmus A5 beschreibt

dabei die zufallsbasierte Auswahl einer Lösung aus der k-Nachbarschaft in Bezug auf die Prioritätsliste einer Lösung.

Algorithmus A5: Nachbarschaft-Prioritätsliste – $random\_priority\_neighbor(y, k)$

Input: y … Ursprungslösung

k … Weite der Nachbarschaft

1. $J \leftarrow$ Anzahl Aktivitäten in y

2. $y' \leftarrow y$

3. for i in $1, 2, \dots, k$ do

$y' \leftarrow insertion\_move(y')$

4. return y'

Ein ähnliches Vorgehen wird in Algorithmus A6 dargestellt. Dieser beschreibt die zufallsbasierte Wahl einer Lösung aus der k-Nachbarschaft in Bezug auf die Modusliste der Ursprungslösung.

Algorithmus A6: Nachbarschaft-Modusliste – $random\_mode\_neighbor(y, k)$

Input: y … Ursprungslösung

k … Weite der Nachbarschaft

1. $J \leftarrow$ Anzahl Aktivitäten in y

2. $y' \leftarrow y$

3. for i in $1, 2, \dots, k$ do

$y' \leftarrow mode\_flip\_move(y')$

4. return y'

Die Algorithmen A5 und A6 können nun in der VNS dazu verwendet werden, Lösungen aus einer beliebigen k-Nachbarschaft auszuwählen. Dies wird dazu verwendet, um ein lokales Optimum wieder verlassen zu können. Dazu müssen diese lokalen Optima erst einmal gefunden werden, was in der VNS durch eine lokale Suche (Algorithmus A7) um die aktuell beste Lösung realisiert wird. Dabei wird die in Abschnitt 2.1.2.1 beschriebene unmittelbare Nachbarschaft $N_0(y)$ wie dort beschrieben konstruiert und vollständig in Bezug auf die in Abschnitt 1.1.6 beschriebene Zielfunktion $g(\cdot)$ evaluiert.

Algorithmus A7: Lokale Suche – $local\_search(y)$

Input: y … Ursprungslösung

Parameter zur Erstellung von $N_0(y)$

Parameter zur Berechnung der Zielfunktion $g(\cdot)$

1. $y_{best} \leftarrow y$

2. Erstelle Nachbarschaft $N_0(y)$

3. for all $y' \in N_0(y)$

   3.1 evaluate $g(y')$

   3.2 if $g(y') < g(y_{best})$ then $y_{best} \leftarrow y'$

4. return y_{best}

Es ist zu beachten, dass für die gegebene Problemstellung sowohl die Konstruktion der Nachbarschaft $N_0(y)$ als auch die Berechnung der Zielfunktion einigermaßen komplexe Vorgänge darstellen. Da an dieser Stelle der Fokus auf der Umsetzung der VNS liegt, und in den jeweiligen Kapiteln die Vorgehensweise für die Konstruktion bzw. Berechnung bereits detailliert dargestellt sind, wird hier auf die algorithmische Darstellung dieser Schritte verzichtet.

Die lokale Suche wird nun in der nach Hansen et al (2010) als *best improvement* bezeichneten Vorgehensweise iterativ angewandt, um zu einem lokalen Optimum zu gelangen. Algorithmus A8 gibt das Vorgehen wieder.

Algorithmus A8: Iterative lokale Suche (*best improvement*) – $best\_improvement(y)$

Input: y … Ursprungslösung

Inputs für $local\_search()$

1. $y_{best} \leftarrow y$
2. while $stop = false$ do
   - 2.1 $y_{new} \leftarrow local\_search(y_{best})$
   - 2.2 if $y_{new} = y_{best}$ then $stop \leftarrow true$

     else $y_{best} \leftarrow y_{new}$
3. return y_{best}

Zu beachten ist dabei, dass der Algorithmus A8 so lange läuft, bis in der unmittelbaren Nachbarschaft der aktuell besten Lösung keine bessere Lösung mehr durch $local\_search()$ gefunden wird. Somit ist das Ergebnis ein lokales Optimum im Sinne dieser Umgebung (Hansen et al 2010). Um das lokale Optimum nun auch wieder verlassen zu können, falls im Lösungsraum noch bessere Lösungen existieren, wird das bereits erwähnte *shaking* angewandt. Dabei wird die lokale Suche auf Lösungen einer erweiterten Nachbarschaft der aktuell besten Lösung angewandt, wo sich eventuell eine noch bessere Lösung findet. Algorithmus A9 beschreibt das Vorgehen, wobei bereits die in dieser Arbeit verwendete Lösungsstruktur und die in Abschnitt 2.1.2.2 beschriebene k-Nachbarschaft berücksichtigt wird. Der *shake step* entspricht dabei der zufallsbasierten Wahl einer Lösung aus $N^{(k)}(y)$.

Algorithmus A9: *shaking* – $shake(y, k)$

Input: y … Ursprungslösung

k … Weite der Nachbarschaft

1. Wähle $k' \in \{0, 1, \dots, k\}$ zufällig
2. $k'' \leftarrow k - k'$
3. $y' \leftarrow random\_priority\_neighbor(y, k')$

4. $y'' \leftarrow random\_mode\_neighbor(y', k'')$

5. return y''

Mit diesen Algorithmen bzw. Funktionen sind nun alle Elemente des VNS-Algorithmus soweit definiert, dass das eigentliche Vorgehen sparsam beschrieben werden kann. Algorithmus A10 gibt das Vorgehen der VNS wieder. Neben geeigneten Stop-Kriterien ist als einziger Parameter ist die maximale Weite der k-Nachbarschaft zu wählen, aus der im *shake step* Lösungen gesucht werden.

Algorithmus A10: Variable Neighborhood Search (VNS) – $VNS(k_{max}, stop\_crit)$

Input: k_{max} … maximale Weite der Nachbarschaft im *shake step*

$stop\_crit$… Stop-Kriterien für die Kontrolle der Laufzeit bzw. des Rechenaufwands

1. Wähle Ursprungslösung y zufällig aus dem Lösungsraum

2. $k \leftarrow 1$

3. $y_{best} \leftarrow best\_improvement(y)$

4. while $stop = false$ do

   4.1 $y' \leftarrow shake(y_{best}, k)$

   4.2 $y'' \leftarrow local\_search(y')$

   4.3 if $g(y'') < g(y_{best})$ do

   $y_{best} \leftarrow best\_improvement(y'')$

   $k \leftarrow 1$

   else

   $k \leftarrow min\{k + 1, k_{max}\}$

   4.4 if "Stop-Kriterium aus $stop\_crit$ erfüllt" do

   $stop \leftarrow true$

5. return y_{best}

2.1.4 Parameter im VNS-Algorithmus

Für die Umsetzung des VNS auf die ausgewählten Problemstellungen werden die in Tabelle 6 angegebenen Parameter und Stop-Kriterien verwendet.

Tabelle 6: Parametrisierung der VNS-Algorithmus

$k_{max} = 100$	Die Wahl eines sehr hohen *k* soll gewährleisten, dass in einem großen Bereich um die aktuell beste Lösung nach besseren Lösungen im ***shake step*** gesucht wird.
Stop-Kriterien	Maximale Anzahl an berechneten Lösungen: $N_{max} > 20000$ Anzahl an Iterationen ohne Verbesserung der Lösung falls maximal zulässiges k erreicht ist: $N_{kmax} > 20$

Änderung der Konstanten für den Strafterm

Eine weitere Überlegung betrifft die Veränderung der Konstanten $c_1, \dots, c_N$, die das Ausmaß der Bestrafung einer Ressourcenüberschreitung bei den nicht-erneuerbaren Ressourcen steuern. Wie in Abschnitt 1.1.6 angesprochen, werden diese Konstanten nicht fix gewählt. Die Konstanten werden im Verlauf des Algorithmus so abgeändert, dass diese monoton mit der Laufzeit wachsen. Somit werden anfangs unzulässige Lösungen gar nicht oder nur mäßig pönalisiert, gegen Ende der Laufzeit immer stärker. Die Laufzeit wird dabei durch die Anzahl an bereits berechneten Lösungen festgelegt, da diese auch eines der Stop-Kriterien darstellt.

Mit den folgenden Inputs

N_{iter} … Anzahl der bis zur bisherigen Iteration berechneten Lösungen

N_{max}… Anzahl maximal zu berechnender Lösungen

$$w_{iter} = \left\lceil 1000 \cdot \frac{N_{iter}}{N_{max}} \right\rceil$$

Berechnen sich die Konstanten im Schritt $iter$ durch

$$c_n(iter) = \exp\left(\frac{w_{iter}}{25} - 30\right) \cdot \mathbb{1}_{\{w_{iter} \geq 500\}}$$

Die Berechnung ist so angepasst, dass sich eine vergleichbare Steigerung im Verlauf des Algorithmus ergibt, wie im später beschriebenen GA. Ab der Hälfte der Laufzeit beginnt die Konstante positiv zu sein, und schlagend zu werden. Ab ca. 80 % der Laufzeit überschreiten die Kosten für eine Ressourcenüberschreitung die gängigen *makespans*, sodass ab dann zulässige Lösungen in jedem Fall als vorteilhaft gegenüber unzulässigen Lösungen.

2.2 Genetischer Algorithmus

2.2.1 Grundidee

Ein genetischer Algorithmus (GA) ist eine weit verbreitete heuristische Methode zur Optimierung und geht auf Holland (1975) zurück. Basis ist eine Population von Lösungen, welche schrittweise in Generationen verändert wird. Dabei werden Mechanismen der natürlichen Evolution nachgeahmt, um relevante Informationen von einer Generation zur nächsten weiterzugeben und die Güte der Population von einer Generation zur nächsten zu verbessern (Whitley, 1994). Eingesetzte Mechanismen sind dabei (1) Selektion, (2) Paarung (*cross over*) und (3) Mutation.

Die hier verwendeten Mechanismen werden im Folgenden beschrieben. Zunächst sollen wesentliche Elemente und Begriffe des Algorithmus beschrieben werden.

2.2.2 Begriffsdefinitionen

Individuum

Ein Individuum ist eine einzelne Lösung y der zu optimierenden Problemstellung. Im vorliegenden Problem handelt es sich dabei wie früher beschrieben um ein Paar einer Prioritätsliste P und einer Modusliste M.

Population

Eine Menge an Individuen wird als Population bezeichnet. Die Populationsgröße N_{pop} (d.h. die Anzahl an Individuen in der Population) ist dabei ein relevanter Parameter des GA.

Generation

Die Population wird über die Zeit in diskreten Schritten verändert, indem auf die Individuen einer Population bestimmte Mechanismen angewandt werden, welche letztendlich zu einer neuen Population (der gleichen Größe) führen. Diese über die Zeit veränderten Populationen werden als Generationen bezeichnet. Die Anzahl an maximal betrachteten Generationen ist ein weiterer Parameter des GA und beschränkt im Wesentlichen die Laufzeit.

Fitness

Die Fitness eines Individuums bezeichnet die Güte der Lösung. Die Fitness ist dabei eine Funktion der Zielfunktion. Da es sich im vorliegenden Fall um ein Minimierungsproblem handelt, geht eine niedrigere Zielfunktion mit höherer Fitness einher. Fitness kann zwischen Individuen verglichen werden. Das fittere Individuum ist dabei dasjenige, mit dem niedrigeren Zielfunktionswert (für die hier betrachtete Problemstellung).

Aufbauend auf diesen Aspekten werden im Folgenden die evolutionären Mechanismen beschrieben, die im hier verwendeten Algorithmus angewendet werden.

2.2.3 Evolutionäre Mechanismen

2.2.3.1 Elitismus

Dieser Mechanismus wählt einen (kleinen) Anteil der fittesten Individuen aus, und übernimmt diese automatisch und unverändert in die nächste Generation. Somit ist garantiert, dass die Informationen der aktuell fittesten Individuen in der nächsten Generation erhalten bleiben. Algorithmus A11 beschreibt dieses Vorgehen.

Algorithmus A11: Elitismus – $elite\_choice(pop,\ n_{elite})$

Input: $pop = \left\{y_1, y_2, \dots, y_{N_{pop}}\right\}$ … Population

n_{elite} … Anzahl an gewählten Individuen

1. Sortiere die Population absteigend nach Fitness (aufsteigend nach der Zielfunktion)

$y'_1, y'_2, \dots, y'_{N_{pop}}$, sodass $g(y'_1) \leq g(y'_2) \leq \cdots \leq g\left(y'_{N_{pop}}\right)$

3. Gib die n_{elite} besten Individuen zurück

return $\{y'_1, y'_2, \dots, y'_{n_{elite}}\}$

2.2.3.2 Selektion

Der Selektionsmechanismus baut darauf auf, dass Individuen in Konkurrenz zueinander stehen. Dabei haben Individuen mit einer höheren Fitness eine höhere Chance, von einer Generation zur nächsten übernommen zu werden bzw. zur Paarung mit anderen Individuen zu gelangen.

Es wird dazu die Methode der *tournament selection* (Razali & Geraghty, 2011) verwendet. Dabei werden zufällig Paare von Individuen aus der Population gewählt, und in Bezug auf deren Fitness verglichen. Das fittere Individuum des Paares wird in die nächste Generation übernommen. Das Vorgehen wird so lange wiederholt, bis die gewünschte Anzahl an

Individuen erreicht ist. Die Ziehung erfolgt dabei jedes Mal auf Basis der gesamten Population in der aktuellen Generation (d.h. mit Zurücklegen). Algorithmus A12 beschreibt das Vorgehen.

Algorithmus A12: Tournament selection – $tournament\_selection(pop, n_{out})$

Input: pop … Population

n_{out}

1. $pop_{new} \leftarrow \{\}$

2. while $\#pop_{new} < n_{out}$ do

    2.1 Wähle $y, y' \in pop$ zufällig

    2.2 if $g(y) < g(y')$ do

        $pop_{new} \leftarrow pop_{new} \cup \{y\}$

    else

        $pop_{new} \leftarrow pop_{new} \cup \{y'\}$

3. return pop_{new}

2.2.3.3 Paarung (*crossover*)

Hier werden zufällig Paare von Individuen aus der vorher selektierten Population gewählt, welche dann genetische Information austauschen. Üblicherweise wird nicht die gesamte Population weitergepaart, sondern nur ein bestimmter Anteil der Individuen. Beim Prozess der Paarung wird das Prinzip des *crossover* (Whitley, 1994) verwendet. Die genetische Information der Eltern-Individuen wird dabei in Stücke zerteilt. Diese werden so wieder zusammengesetzt, dass die Teile von den beiden Eltern stammen und zusammen die Information eines neuen Individuums bilden. Es entstehen auf diese Art zwei Nachkommen, die jeweils teilweise die Information mit beiden Eltern teilen. Die zweiteiligen Lösungen, die für das hier behandelte Problem nötig sind, brauchen eine Kombination aus zwei *crossover*-Mechanismen. Ein Mechanismus bezieht sich die Prioritätslisten, der andere auf die Moduslisten.

Crossover der Prioritätslisten

Da es sich bei der Prioritätsliste um eine Permutation der Aktivitäten handelt, trägt die Ordnung der Elemente die wesentliche Information. Die Prioritätsliste ist außerdem nicht zirkulär, d.h. die erste und letzte Position werden nicht als verbunden betrachtet. Daher wird das *linear order crossover*, kurz LOX (z. B. Falkenauer & Bouffouix, 1991; Liaw, 2000) verwendet. Das Vorgehen wird in Algorithmus A13 beschrieben.

Algorithmus A13: *Linear order crossover* – $lox\_crossover(y, y')$

Input: $y = (P, M)$ … erstes Eltern-Individuum mit J … Anzahl an Aktivitäten in M und P

$y' = (P', M')$ … zweites Eltern-Individuum

mit $P = (p_1, p_2, \dots, p_J)$ … Prioritätsliste des ersten Eltern-Individuums

$P' = (p'_1, p'_2, \dots, p'_J)$ … Prioritätsliste des zweiten Eltern-Individuums

1. Wähle *cut points* c, c' zufällig

$c \in \{1, \dots, J-1\}, c' \in \{c+1, \dots, J\}$

$k = c' - c + 1$

2. Schneide die Sequenz $S = (p_c, \dots, p_{c'})$ aus P heraus

3. Finde Indizes $I = \{i_1, \dots, i_k\}$, sodass $p'_i \notin S$ für alle $i \in I$

4. Erstelle eine neue Prioritätsliste $P_{new} = \left(p_1^{new}, \dots, p_J^{new}\right)$

4.1 Setze Positionen $\left(p_c^{new}, \dots, p_{c'}^{new}\right)$ als $(p_c, \dots, p_{c'})$

4.2 Setze Positionen $\left(p_1^{new}, \dots, p_{c-1}^{new}, p_{c'+1}^{new}, \dots, p_J^{new}\right)$ als $\left(p'_{i_1}, \dots, p'_{i_k}\right)$

5. Wiederhole Schritte 2-4 mit vertauschten Rollen von P und P' und erstelle so P'_{new}

6. Erstelle neue Lösungen

$y_{new} = replace\_P(y, P_{new})$

$y'_{new} = replace\_P(y', P'_{new})$

7. Gib die beiden neuen Lösungen zurück

return (y_{new}, y'_{new})

Tabelle 7 illustriert die Anwendung der einzelnen Schritte des Algorithmus A13 für ein Beispiel von zwei Prioritätslisten zweier Eltern-Individuen.

Tabelle 7: Beispiel der Konstruktion einer neuen Prioritätsliste aus zwei Eltern-Individuen mittels LOX

Inputs	Aktivitäten: $J = 8$
	$P = (8, 5, 6, 1, 3, 4, 7, 2)$
	$P' = (4, 6, 5, 1, 8, 2, 7, 3)$
Cutpoints	$c = 3, c' = 6, k = 6 - 3 + 1 = 4$
$S = (p_c, \dots, p_{c'})$	$S = (6, 1, 3, 4)$
Indizes I	$i_1 = 3, i_2 = 5, i_3 = 6, i_4 = 7$
$(\mathrm{p'}_{\mathrm{i}_1}, \dots, \mathrm{p'}_{\mathrm{i}_k})$	$(5, 8, 2, 7)$
P_{new}	$(5, 8, 6, 1, 3, 4, 2, 7)$

Das in Tabelle 7 realisierte Ergebnis des *crossovers* ist in Abbildung 6 in Anlehnung an die Darstellung in Liaw (2000) noch einmal illustriert.

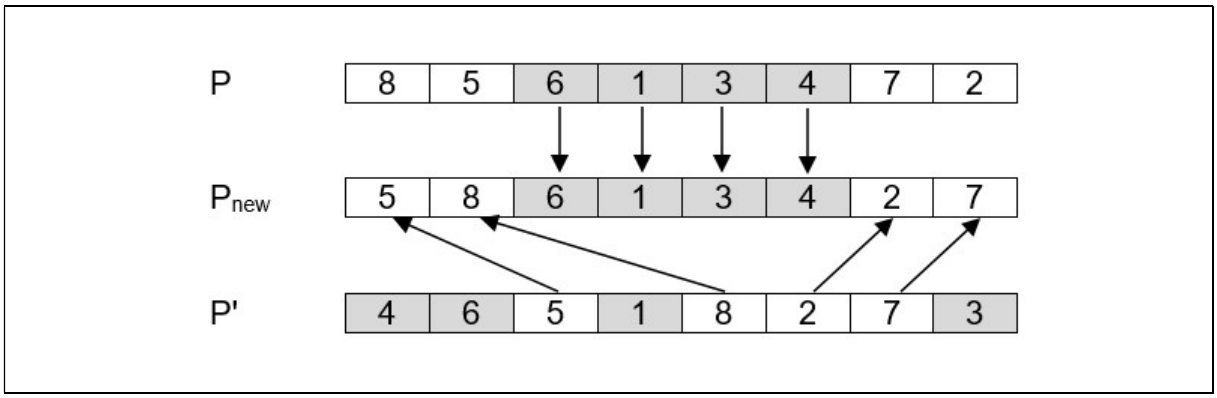

Abbildung 6: Illustration des in Tabelle 7 beschriebenen Beispiels zum LOX in Anlehnung an Liaw (2000).

Crossover der Moduslisten

Bei der Modusliste liegt die Information nicht in der Ordnung der Elemente, sondern in den Einträgen an den J Positionen selbst. Daher wird eine Version des *2-point-crossover* (z. B. Umbarkar & Sheth, 2015) angewandt. Häufig wird dieses *crossover* für Lösungen angewandt, welche auf einer binären Repräsentation basieren. In der Prioritätsliste können die Einträge für die Einzelnen Aktivitäten allerdings mehr als zwei (nämlich M_j) Zustände aufweisen. Trotzdem lässt sich das Prinzip leicht auf diese Situation übertragen. Das Vorgehen ist in Algorithmus A14 dargestellt.

Algorithmus A14: *2-point-crossover* – $two\_point\_crossover(y, y')$

Input: $y = (P, M)$ … erstes Eltern-Individuum mit J … Anzahl an Aktivitäten in M und P

$y' = (P', M')$ … zweites Eltern-Individuum

mit $M = (m_1, m_2, \dots, m_J)$ … Modusliste des ersten Eltern-Individuums

$M' = (m'_1, m'_2, \dots, m'_J)$ … Modusliste des zweiten Eltern-Individuums

1. Wähle *cutpoints* c, c' zufällig

$c \in \{1, \dots, J-1\}, c' \in \{c+1, \dots, J\}$,

2. Schneide die Sequenz $S = (m_c, \dots, m_{c'})$ aus M heraus

3. Schneide die Sequenz $S' = (m'_c, \dots, m'_{c'})$ aus M' heraus

4. Erstelle eine neue Modusliste $M_{new} = M$

4.1 Ersetze Einträge $= (m_c, \dots, m_{c'})$ M_{new} durch S'

5. Erstelle eine neue Modusliste $M'_{new} = M'$

5.1 Ersetze Einträge $(m'_c, \dots, m'_{c'})$ in M'_{new} durch S

6. Erstelle neue Lösungen

$y_{new} = replace\_M(y, M_{new})$

$y'_{new} = replace\_M(y', M'_{new})$

7. Gib die beiden neuen Moduslisten zurück

return (y_{new}, y'_{new})

Tabelle 8 illustriert das in Algorithmus A14 dargestellte Vorgehen für am Beispiel zweier Moduslisten.

Tabelle 8: Beispiel der Konstruktion einer neuen Modusliste aus zwei Eltern-Individuen mittels dem adaptierten *2-point-crossover*

Inputs	Aktivitäten: $J = 8$
	$M = (1, 2, 3, 1, 3, 1, 1, 1)$
	$M' = (1, 3, 2, 1, 2, 2, 3, 1)$
Cutpoints:	$c = 3, c' = 6$
Entfernte Sequenzen	$S = (3, 1, 3, 1)$
	$S' = (2, 1, 2, 2)$
Neue Moduslisten	$M_{new} = (1, 2, 2, 1, 2, 2, 1, 1)$
	$M'_{new} = (1, 3, 3, 1, 3, 1, 3, 1)$

Abbildung 7 illustriert das in Tabelle 8 beschriebene Beispiel noch einmal grafisch.

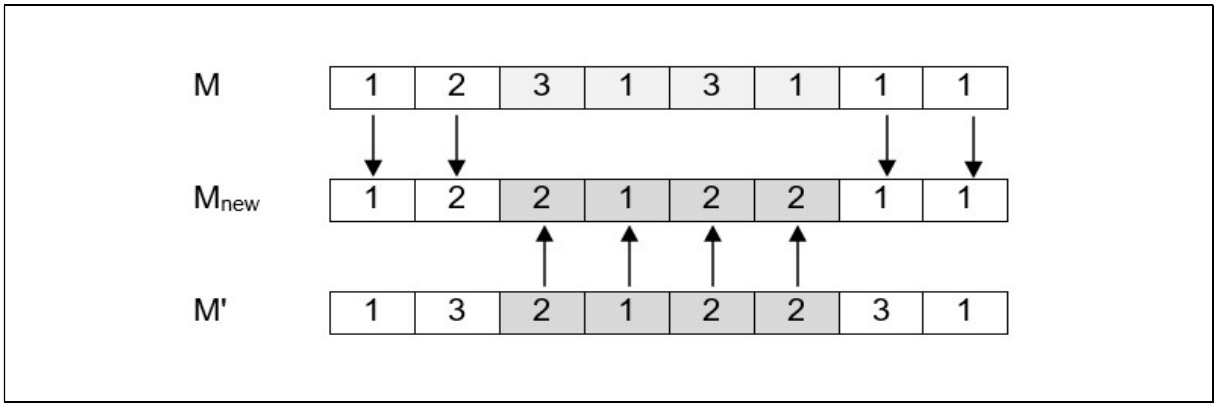

Abbildung 7: Illustration des in Tabelle 8 beschriebenen Beispiels zum *2-point-crossover.*

2.2.3.4 Mutation

Eine Mutation beschreibt eine (geringe) Änderung der genetischen Information eines einzelnen Individuums (Liaw, 2000). Wiederum müssen aufgrund der zweiteiligen Lösung unterschiedliche Mechanismen der Mutation auf diese beiden Teile angewandt werden.

Mutation der Prioritätsliste

Auf die Prioritätsliste wird die Methode der *swap mutation* (Liaw, 2000) angewandt. Dabei werden zwei zufällig ausgewählte Positionen vertauscht. Algorithmus A15 beschreibt das Vorgehen.

Algorithmus A15: *swap mutation* – $swap\_mutation(y)$

Input: $y = (P, M)$ … Lösung mit J … Anzahl an Aktivitäten in M und P

1. Wähle *swap points* c, c' zufällig

   $c \in \{1, \dots, J-1\},\ c' \in \{c+1, \dots, J\}$

2. Adaptiere die Lösung

   2.1 Erstelle eine neue Prioritätsliste $P_{new} = P$

   2.2 Ersetze in P_{new} die Einträge $(p_c,\ p_{c'})$ durch$(p_{c'}, p_c)$

   2.3 Erstelle eine neue Lösung $y_{new} = replace\_P(y, P_{new})$

3. return (y_{new})

Abbildung 8 illustriert dieses Vorgehen am Beispiel einer Prioritätsliste.

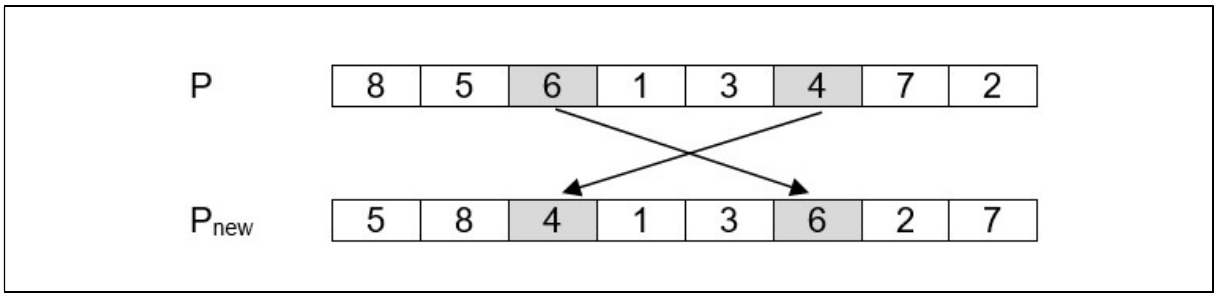

Abbildung 8: Beispiel des in Algorithmus A15 beschriebenen *swap mutation.*

Mutation der Modusliste

Für die Modusliste wird ein *mode flip* als Mutationsmechanismus gewählt. Dabei wird der Modus einer zufällig gewählten Aktivität durch einen der möglichen Modi der entsprechenden Aktivität ersetzt. Algorithmus A16 stellt das Vorgehen dar.

Algorithmus A16*: flip mutation* – $flip\_mutation(y, M_{possible})$

Input: $y = (P, M)$ … Lösung mit J … Anzahl an Aktivitäten in M und P

$M_{possible} = \{M_1, \dots, M_J\}$ …Mögliche Ausführungsmodi der Aktivitäten

1. Wähle *mutation point* c zufällig

$c \in \{1, \dots, J\}$

2. Wähle zufällig einen neuen Modus für diese Aktivität

$m'_c \in \{1, \dots, M_c\}$

3. Adaptiere die Lösung

3.1 Erstelle eine neue Modusliste $M_{new} = M$

3.2 Ersetze in M_{new} den Eintrag m_c durch m'_c

3.3 Erstelle eine neue Lösung $y_{new} = replace\_M(y, M_{new})$

5. return (y_{new})

Zu beachten ist dabei, dass prinzipiell auch der ursprüngliche Modus zufällig als neuer Modus gezogen werden kann. In diesem Fall hat sich die Information nicht durch die Mutation geändert. Da die Mutation nur mit einer bestimmten Wahrscheinlichkeit bzw. nur bei einem bestimmten Anteil der Population eingesetzt wird, reduziert dieses Vorgehen lediglich diese vorher festgelegte Wahrscheinlichkeit bzw. den Anteil.

2.2.4 Umsetzung des GA

Die beschriebenen Prinzipien von Elitismus, Selektion, Crossover und Mutation werden nun zu einem genetischen Algorithmus vereint. Dabei werden die Mechanismen *crossover* und Mutation in jeder Generation nur auf einen bestimmten Anteil der Population angewandt. Dies kann so interpretiert werden, dass diese Mechanismen nur mit einer bestimmten Wahrscheinlichkeit eintreten. Der GA ist in Algorithmus A17 dargestellt.

Algorithmus A17: Genetischer Algorithmus

Parameter: N_{pop} … Populationsgröße

p_{cross} … Wahrscheinlichkeit eines *crossovers*

p_{muta} … Wahrscheinlichkeit einer Mutation

p_{elite} … Anteil des Elitismus

$iter_{max}$ … Anzahl an Generationen (Laufzeit)

1. Setze Parameter

$n_{elite} \leftarrow \lceil N_{pop} \cdot p_{elite} \rceil$

$n_{cross} \leftarrow \lceil N_{pop} \cdot p_{cross} \rceil$

$n_{muta} \leftarrow \lceil N_{pop} \cdot p_{muta} \rceil$

$n_{sel} \leftarrow N_{pop} - n_{elite}$

$iter \leftarrow 0$

2. Erstelle Startpopulation pop_0 zufällig

3. while $iter \leq iter_{max}$ do

3.1 Elitismus

$pop_{elite} \leftarrow elite\_choice(pop_{iter}, n_{elite})$

3.2 Selektion

$pop_{sel} \leftarrow tournament\_selection(pop_{iter}, n_{sel})$

3.3 Crossover

3.3.1 Crossover Prioritätsliste

$k \leftarrow 0$

$sel_{cross} \leftarrow \{\}$

while $k < n_{cross}$ do

wähle zufällig $i, i' \in \{1, \dots, n_{sel}\} \setminus sel_{cross}, i \neq i'$

Ersetze y_i und $y_{i'}$ in pop_{sel} durch $lox\_crossover(y_i, y_{i'})$

$sel_{cross} \leftarrow sel_{cross} \cup \{i, i'\}$

$k \leftarrow k + 2$

3.3.2 Crossover Modusliste

$k \leftarrow 0$

$sel_{cross} \leftarrow \{\}$

while $k < n_{cross}$ do

wähle zufällig $i, i' \in \{1, \dots, n_{sel}\} \setminus sel_{cross}, i \neq i'$

Ersetze y_i und $y_{i'}$ in pop_{sel} durch $two\_point\_crossover(y_i, y_{i'})$

$sel_{cross} \leftarrow sel_{cross} \cup \{i, i'\}$

$k \leftarrow k + 2$

3.4 Mutation

3.4.1 Mutation Prioritätsliste

$k \leftarrow 0$

$sel_{muta} \leftarrow \{\}$

while $k < n_{muta}$ do

wähle zufällig $i \in \{1, \dots, n_{sel}\} \setminus sel_{muta}$

Ersetze y_i in pop_{sel} durch $swap\_mutation(y_i)$

$sel_{muta} \leftarrow sel_{muta} \cup \{i\}$

$k \leftarrow k + 1$

3.4.2 Modusliste

$k \leftarrow 0$

$sel_{muta} \leftarrow \{\}$

while $k < n_{muta}$ do

wähle zufällig $i \in \{1, \dots, n_{sel}\} \setminus sel_{muta}$

Ersetze y_i in pop_{sel} durch $flip\_mutation(y_i, M_{possible})$

$sel_{muta} \leftarrow sel_{muta} \cup \{i\}$

$k \leftarrow k + 1$

3.5. Adaptiere die Population

$iter \leftarrow iter + 1$

$pop_{iter} \leftarrow pop_{elite} \cup pop_{sel}$

3.6. Adaptiere y_{best}

3.6.1 Finde beste Lösung der aktuellen Population

y'_{best}, sodass $\forall\, y \in pop_{iter}$: $g(y'_{best}) \leq g(y)$

3.6.2 if $g(y'_{best}) < g(y_{best})$ do

$y_{best} \leftarrow y'_{best}$

4. return y_{best}

2.2.5 Parameter im GA-Algorithmus

Für die Anwendung des GA auf die ausgewählten Problemstellungen wird die in Tabelle 9 angegebene Parametrisierung verwendet.

Tabelle 9: Parametrisierung des GA

Parameter	Beschreibung
$N_{pop} = 100$	Anzahl an Individuen in der Population
$p_{cross} = 0.2$	Wahrscheinlichkeit eines crossovers Wird für beide crossover-Prozesse unabhängig und hintereinander angewandt. → Die Wahrscheinlichkeit, dass ein Individuum der Population bei mindestens einem der beiden Lösungsteile ein crossover erfährt, beträgt demnach 0.36
$p_{muta} = 0.02$	Wahrscheinlichkeit einer Mutation Wird für beide Mutations-Prozesse unabhängig und hintereinander angewandt → Die Wahrscheinlichkeit, dass ein Individuum der Population bei mindestens einem der beiden Lösungsteile eine Mutation erfährt, beträgt demnach 0.036
$p_{elite} = 0.02$	Anteil des Elitismus
$iter_{max} = 1000$	Anzahl an Generationen (Laufzeit)

Änderung der Konstanten für den Strafterm

Wie bereits im VNS-Algorithmus wird auch im GA die Zielfunktion im Verlauf des Algorithmus immer stärker von einer etwaigen Überschreitung der nicht-erneuerbaren Ressourcen einer Lösung beeinflusst. Die Veränderung der Konstanten folgt dabei folgender Funktion

Berechnen sich die Konstanten im Schritt $iter$ durch

$$c_n(iter) = \exp\left(\frac{iter}{25} - 30\right) \cdot \mathbb{1}_{\{iter \geq 500\}}$$

Da die Anzahl an Generationen auf 1000 beschränkt wird, ergibt sich wiederum ab der Hälfte der Laufzeit eine positive Konstante und ab ca. 80 % der Laufzeit werden unzulässige Lösungen praktisch immer von zulässigen Lösungen übertrumpft (siehe Abschnitt 2.1.4).

2.3 Computationale Umsetzung

Die Umsetzung der oben beschriebenen Algorithmen und Funktionen erfolgt in R (R Core Team, 2017). Abschnitt 6.5 im Appendix enthält Auszüge des R-Codes für einige ausgewählte Funktionen. Die Evaluierung der Zielfunktion erfolgt parallelisiert, d.h. es wird die Berechnung der Zielfunktion parallel für die gesamte Population (GA) bzw. die aktuell im Fokus stehende Nachbarschaft (VNS) durchgeführt.

Die Ausführung erfolgte auf einem System mit folgender CPU: 2 x Intel E5-2640 V4 2.40GHz (10 Core) und 128 GB Arbeitsspeicher.

3 Ergebnisse

3.1 Allgemeine Ergebnisse

3.1.1 Gefundene Lösungen

Aufgrund der unterschiedlichen Schwierigkeit der ausgewählten Problemstellungen, der Art der Berücksichtigung der Lösungen und der Konstruktion der Zielfunktion, finden die Algorithmen nicht bei allen Durchläufen zwingend zulässige Lösungen. Zulässig meint in diesem Kontext, dass für keine der nicht-erneuerbaren Ressourcen die Kapazität durch die in der Lösung gewählten Modi überschritten wird. Interessant ist daher, in wie vielen Fällen einer oder beide Algorithmen zu keiner zulässigen Lösung kommen. In 19 Problemstellungen wird von mindestens einem Algorithmus in mindestens einem der drei Durchläufe (drei Zufallsverteilungen der Bearbeitungsdauern) keine zulässige Lösung gefunden. Tabelle 10 fasst die Ergebnisse zusammen.

Tabelle 10: Anzahl an Durchgängen ohne zulässige Lösung

	Keine zulässige Lösung gefunden		
Projektname	VNS	GA	(beide)
j1029_10	0	1	(0)
j1029_7	0	1	(0)
j1634_4	0	1	(0)
j1635_2	0	1	(0)
j1635_3	0	1	(0)
j1637_1	1	0	(0)
j1638_6	1	2	(1)
j166_2	3	3	(3)
j3015_4	1	0	(0)
j303_10	3	3	(3)
j303_6	3	3	(3)
j3034_3	3	3	(3)
j3036_10	3	3	(3)
j3038_2	3	2	(2)
j304_7	3	3	(3)
j3040_7	3	3	(3)
j3048_8	2	0	(0)
j306_7	3	3	(3)
j308_6	3	3	(3)
Summe	35	36	(30)

In fünf (VNS) bzw. vier (GA) Fällen findet nur einer der beiden Algorithmen keine zulässige Lösung. In 30 Fällen finden beide Algorithmen keine. Auffällig ist auch, dass die Anzahl an Projekten, in denen keine zulässige Lösung gefunden wird mit der Anzahl an Aktivitäten steigt. Von jeweils 30 Projekten werden in zwei (10 Aktivitäten), sechs (16 Aktivitäten) und elf (30 Aktivitäten) in zumindest einem Durchgang von zumindest einem Algorithmus keine zulässige Lösung gefunden.

3.1.2 Anteil zulässiger Lösungen pro Projekt

Die Zulässigkeit in der hier berücksichtigten Weise hängt nur von den nicht-erneuerbaren Ressourcen ab. Weiters hängt die Zulässigkeit einer Lösung nur von der Modusliste und nicht von der Prioritätsliste ab. Es liegt daher nahe zu untersuchen, wie viele der möglichen Moduslisten pro Projekt überhaupt zu einer zulässigen Lösung führen. Dieses Problem wird für die Projekte mit $J = 10$ und $J = 16$ Aktivitäten durch vollständige Enumeration gelöst. D.h. es werden für alle betrachteten Projekte alle möglichen Moduslisten erzeugt und hinsichtlich der Zulässigkeit evaluiert. Die Anzahl an Moduslisten entspricht 3^J, da in allen Problemen pro Aktivität drei Modi vorhanden waren. Dies ergibt also 59 049 bzw. 43 046 721 Moduslisten für jedes der 10- bzw. 16-Aktivitäten-Projekte. Für die Projekte mit 30 Aktivitäten übersteigt die Anzahl (pro Projekt $\approx 2.06 \cdot 10^{14}$) an möglichen Moduslisten das Ausmaß, in dem eine vollständige Evaluierung aller Möglichkeiten mit den verfügbaren Ressourcen realisierbar ist.

Abbildung 9 zeigt die zusammengefassten Ergebnisse dieser Analyse.

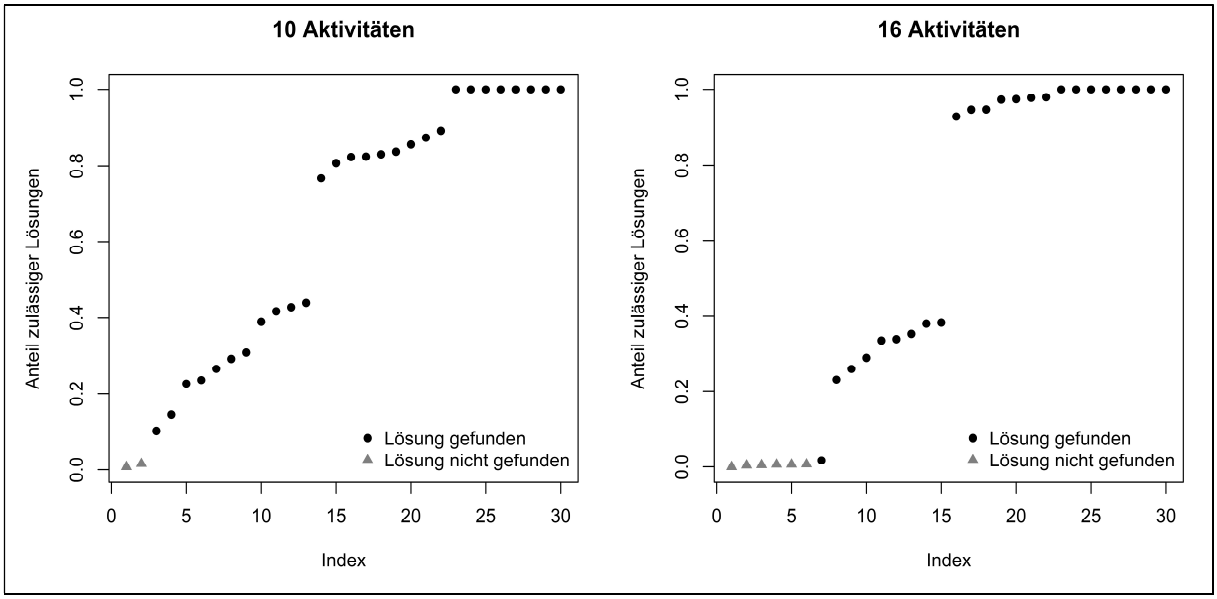

Abbildung 9: Anteil an zulässigen Lösungen (Moduslisten) in den jeweils 30 untersuchten Problemstellungen für 10 und 16 Aktivitäten.

Dabei sind diejenigen Projekte hervorgehoben, die in zumindest einem Durchgang in mindestens einem der Algorithmen keine Lösung hervorgebracht haben. Es ist klar ersichtlich, dass die Probleme beim Auffinden der zulässigen Lösungen mit dem Anteil an generell verfügbaren zulässigen Moduslisten im jeweiligen Projekt zusammenhängen. Der extremste Fall ist Projekt j166_2, wo nur 192 Moduslisten zulässig sind. Das entspricht einem Anteil von nur 0.00045 % zulässigen Lösungen. Bei den Problemen mit zehn Aktivitäten wird das Minimum bei Projekt j1029_7 mit 436 zulässigen Moduslisten erreicht, was einem Anteil von 0.74 % entspricht.

3.2 Vergleich der Algorithmen

3.2.1 Anteil besserer Performance

Ein Hauptinteresse liegt natürlich auch im Vergleich der beiden Algorithmen untereinander. Dazu ist einerseits der Anteil an Problemen interessant, in denen einer der beiden Algorithmen besser abschneidet. Dabei meint besser abschneiden, dass eine beste Lösung gefunden wird, deren Zielfunktionswert besser als bei der besten Lösung des anderen Algorithmus ist. Tabelle 11 fasst diese Ergebnisse zusammen. Zu beachten ist, dass nur mehr diejenigen Projekte berücksichtigt werden, in denen zumindest einer der Algorithmen eine zulässige Lösung findet. Die Ergebnisse sind für die drei untersuchten Modelle der zufälligen Bearbeitungsdauern getrennt dargestellt.

Tabelle 11: Verteilung der besseren gefundenen Lösungen nach Algorithmus und Verteilung

	Bessere zulässige Lösung gefunden					
Verteilung	Lösung nur in GA	GA besser	Gleichwertige Lösung	VNS besser	Lösung nur in VNS	Beide ohne Lösung
Gleich-verteilung	0	10	7	62	1	10
Dreiecks-verteilung	3	7	7	61	2	10
Weibull-verteilung	2	6	8	61	3	10
Summe (%)	5 (2 %)	23 (9 %)	22 (8 %)	184 (68 %)	6 (2 %)	30 (11 %)

Es wird deutlich, dass der VNS dominiert. In 190 Fällen findet der VNS die bessere Lösung bzw. überhaupt eine zulässige Lösung im Vergleich zum GA. Das entspricht knapp 90 % der Fälle, in denen zumindest einer der beiden Algorithmen eine Lösung findet. Als statistischer Test kann für diesen Vergleich ein Binomialtest herangezogen werden. Es werden dazu die nicht entscheidbaren Fälle (gleichwertige Lösung bzw. beide Algorithmen ohne Lösung) eliminiert. Außerdem werden diejenigen Fälle, in denen nur einer der beiden Algorithmen eine Lösung findet, als besser für diesen Algorithmus gezählt. Es ergibt sich somit folgende Ausgangslage: GA besser in 28 Fällen, VNS besser in 190 Fällen.

Getestet wird dann die Nullhypothese, dass die beiden Algorithmen gleichwertig sind, d.h. dass der Anteil an besseren Lösungen für beide Algorithmen gleich 0.5 beträgt.

π ... Wahrscheinlichkeit, dass VNS besser abschneidet

$H_0: \pi = 0.5$

Als Teststatistik T kann die Summe der besseren Lösungen für einen der beiden Algorithmen (hier VNS) herangezogen werden. Unter der H_0 ist diese Teststatistik $B(n, 0.5)$-verteilt. In unserem Fall kann $T = 190$ und $n = 218$ verwendet werden. Mittels Normalverteilungsapproximation erhält man den näherungsweisen z-Wert von 10.07. Es ergibt sich somit ein $p < .001$, d.h. die Nullhypothese der Gleichwertigkeit der Algorithmen kann auf dem Niveau $\alpha = .05$ verworfen werden.

3.2.2 Differenz der Lösungen

Statt der einfachen Entscheidung, welcher Algorithmus besser ist, ist natürlich auch die Höhe des Unterschieds von Interesse. Um das Ausmaß der Differenz zwischen den gefundenen Lösungen zu evaluieren, wird die Differenz in Relation zur schlechteren der beiden gefundenen Lösungen gesetzt. Somit ergibt sich eine relative Abweichung von der schlechteren der beiden Lösungen. Abbildung 10 fasst diese Ergebnisse zusammen. Eine positive Differenz spricht dabei für einen Vorteil beim VNS. Es wird deutlich, dass der VNS teilweise viel besser Lösungen als der GA liefert, in einigen Fällen wird eine Verbesserung der Lösung des GA um mehr als 30 % erreicht. Vereinzelt schneidet jedoch auch der GA um bis zu ca. 18 % besser ab.

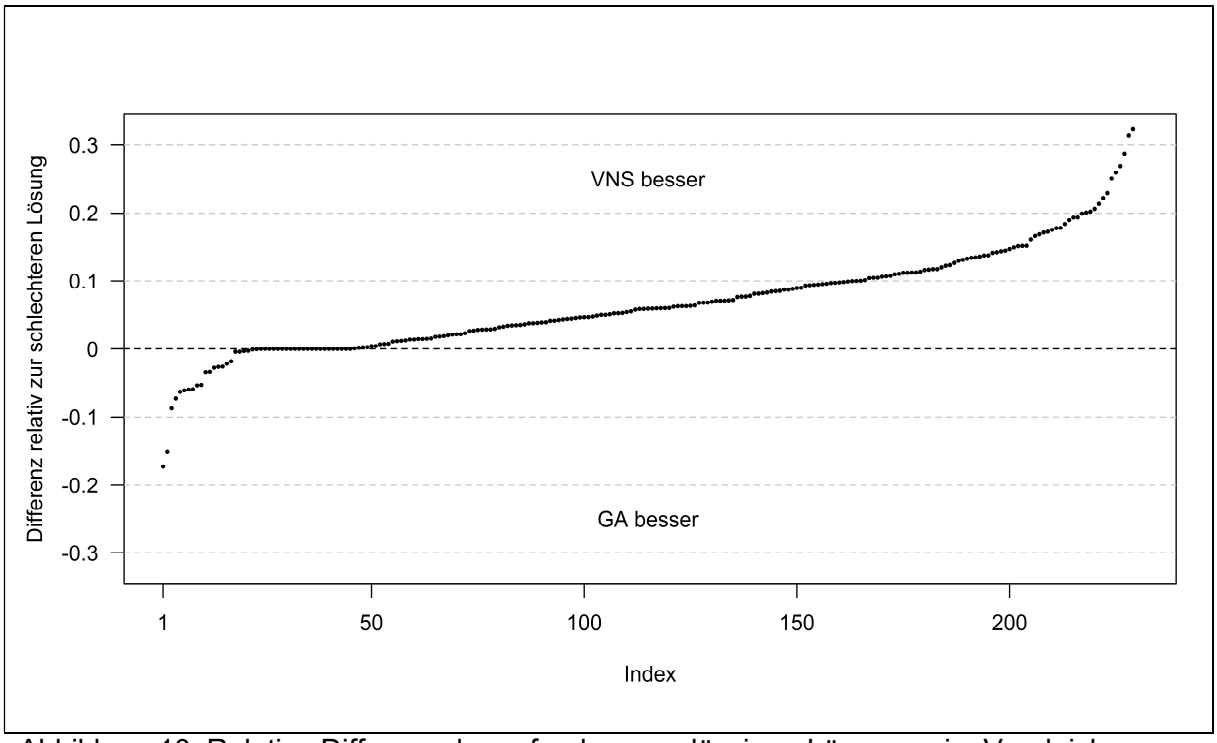

Abbildung 10: Relative Differenz der gefundenen zulässigen Lösungen im Vergleich zwischen GA und VNS

3.3 Value of the stochastic solution

Dem Ansatz der *value of the stochastic solution* von Birge und Louveaux (2011) folgend werden alle Problemstellungen auch lediglich unter Berücksichtigung der deterministischen Bearbeitungszeiten (den Basiszeiten für die Simulation der Szenarien) optimiert. Dazu werden die gleichen Algorithmen und Problemstellungen verwendet. Statt des $AVaR_\alpha$ wird jedoch die simple *makespan* auf Basis der deterministischen Bearbeitungszeiten in die Zielfunktion eingebunden. Die Optimierung erfolgt also auf Basis der Modalwerte (der Basiszeiten B). Birge und Louveaux (2011) empfehlen dafür die Optimierung nach den Erwartungswerten der stochastischen Bearbeitungsdauern. Hier wird trotzdem die Basiszeit B verwendet, auch wenn diese nicht in allen Verteilungen den Erwartungswerten entspricht. Eine Begründung dafür ist, dass die Basiszeiten für die Erstellung der Zufallsvariablen herangezogen werden und per Konstruktion ähnlich den Erwartungswerten sind.

3.3.1 Vergleich mit den ursprünglichen Lösungen

Für die so erhaltenen besten und zulässigen Lösungen wird anschließend die Zielfunktion auf die übliche Weise ($AVaR_\alpha$ auf Basis der in den stochastischen Algorithmen verwendeten Szenarien) berechnet. Somit lassen sich diese Lösungen mit den aus der ursprünglichen Anwendung der Algorithmen erhaltenen Lösungen vergleichen. Die Abbildungen 11 und 12

zeigen die relative Differenz der Lösungen zur jeweils schlechteren Lösungen für alle Probleme, wo in beiden Fällen (Lösung auf Basis der deterministischen Zeiten bzw. auf Basis des $AVaR_\alpha$) eine zulässige Lösung vorhanden ist. Es zeigt sich, dass für beide Algorithmen die Optimierung auf Basis der stochastischen Zielfunktion häufiger zu besseren Ergebnissen führt.

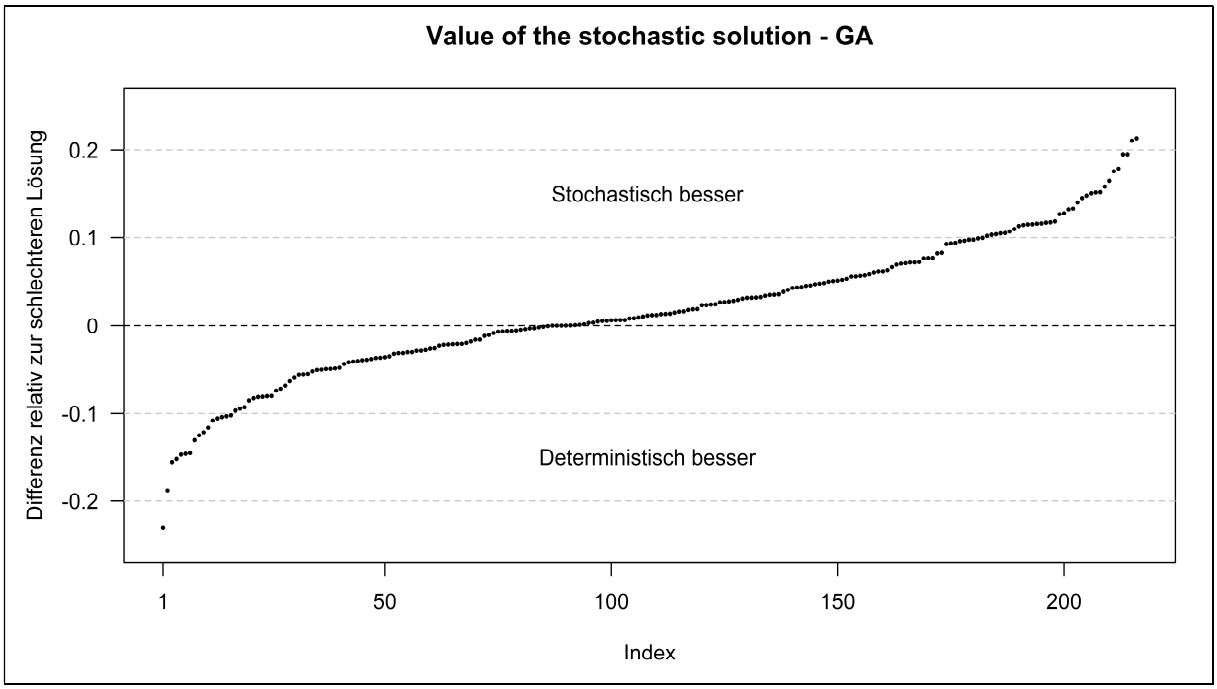

Abbildung 11: Relative Differenz der gefundenen zulässigen Lösungen im Vergleich zwischen stochastisch und deterministisch gefundenen Lösungen im GA

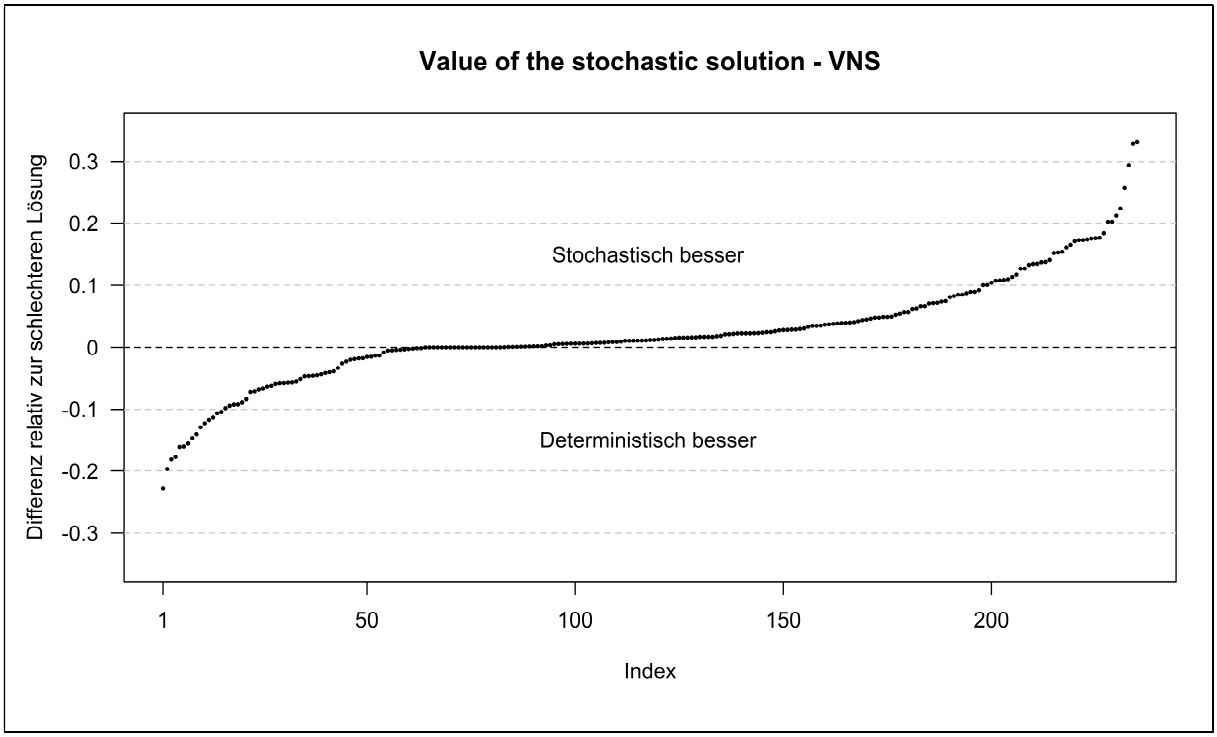

Abbildung 12: Relative Differenz der gefundenen zulässigen Lösungen im Vergleich zwischen stochastisch und deterministisch gefundenen Lösungen im VNS

Tabelle 12 fasst diese Ergebnisse noch einmal zusammen und beinhaltet die p-Werte, welche aus den entsprechenden Binomialtests stammen.

Tabelle 12: Verteilung der besseren gefundenen Lösungen für deterministische bzw. stochastische Zielfunkton

	Bessere Lösung gefunden			
	deterministisch	gleichwertig	stochastisch	p-Wert (*)
GA	86	3	127	.006
VNS	63	15	157	< .001
Beide Algorithmen	61	17	162	< .001

(*) $H_0: \pi = 0.5$ Gleichwertigkeit zwischen deterministisch/stochastisch, unentscheidbare Lösungen werden eliminiert

In den meisten Fällen sind die stochastisch gefundenen Lösungen den deterministisch gefundenen überlegen. Es zeigt sich, dass der Anteil der Projekte in denen in den deterministischen Algorithmen bessere Lösungen gefundenen werden, wider Erwarten sehr hoch ist. Um diesem Phänomen noch auf den Grund zu gehen, wird eine neue Konstruktion der Verteilung der Bearbeitungszeiten untersucht.

3.3.2 Vergleich auf Basis einer neuen Verteilung

Für einen weiteren Vergleich der deterministischen/stochastisch gefundenen Lösungen, wird eine neue Verteilung konstruiert. Diese ist im Appendix (Abschnitt 6.3) beschrieben.

Die Optimierung wird für die neuen Basiszeiten für die 30 Problemstellungen mit $J = 10$ Aktivitäten durchgeführt und mit den Ergebnissen aus den deterministischen Algorithmen von vorher (für diese Probleme) verglichen. Abbildung 13 zeigt die Ergebnisse für den GA.

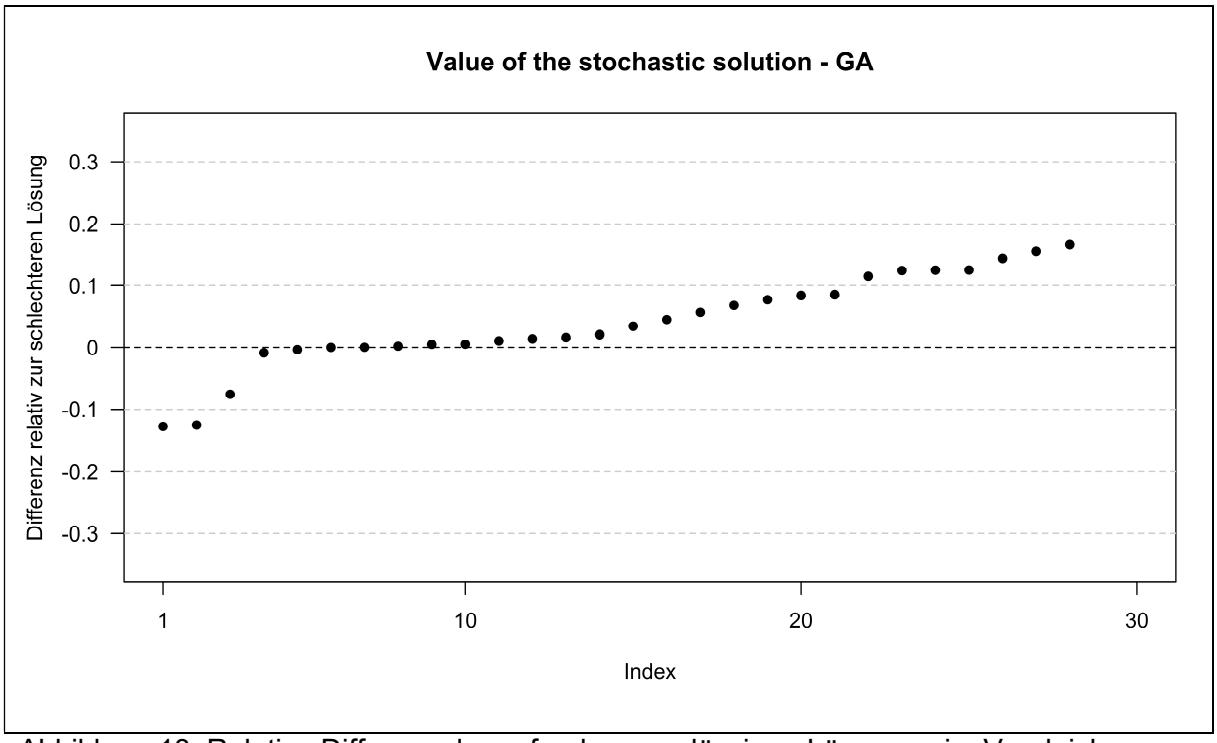

Abbildung 13: Relative Differenz der gefundenen zulässigen Lösungen im Vergleich zwischen stochastisch und deterministisch gefundenen Lösungen im GA. Modellierung nach der neuen Gleichverteilung. Problemstellungen mit $J = 10$ Aktivitäten. (Für zwei Probleme werden bei beiden Methoden keine zulässige Lösung gefunden.)

Es zeigt sich, dass im Vergleich zu den ursprünglich verwendeten Verteilungen der Algorithmus mit der stochastischen Zielfunktion nun etwas deutlicher dem deterministischen Algorithmus überlegen ist. In gleicher Weise wird der Vergleich für den VNS-Algorithmus durchgeführt. Abbildung 14 enthält die entsprechenden Ergebnisse für diesen Algorithmus. Wiederum zeigt sich die Überlegenheit der stochastischen Version des Algorithmus. Tabelle 13 enthält zusätzlich die absoluten Ergebnisse sowie die p-Werte der resultierenden Binomialtests. In allen Fällen zeigen sich die stochastischen Algorithmen als überlegen. Der Fall wo beide Algorithmen betrachtet werden, enthält die jeweils beste Lösung aus den stochastischen Versionen von VNS und GA gegenüber den besten Lösungen aus den deterministischen Versionen der beiden Algorithmen.

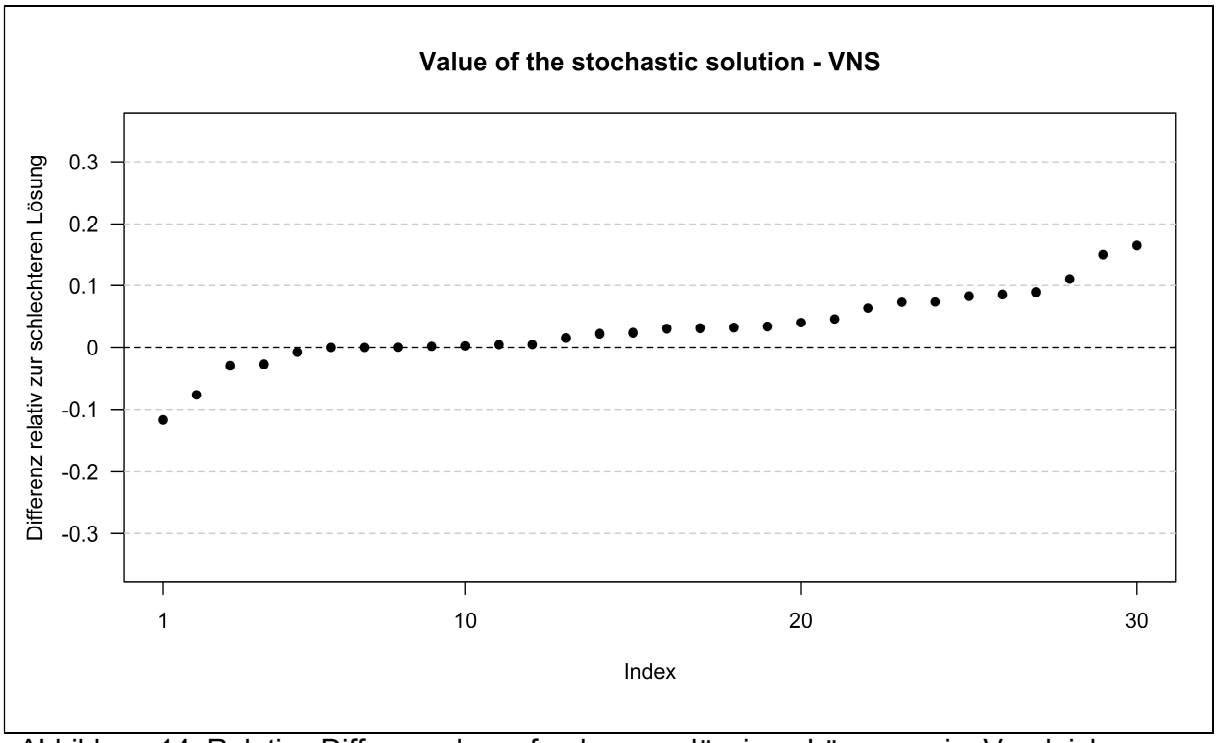

Abbildung 14: Relative Differenz der gefundenen zulässigen Lösungen im Vergleich zwischen stochastisch und deterministisch gefundenen Lösungen im VNS. Modellierung nach der neuen Gleichverteilung. Problemstellungen mit $J = 10$ Aktivitäten.

Tabelle 13: Verteilung der besseren gefundenen Lösungen für deterministische bzw. stochastische Zielfunkton. Modellierung nach der neuen Gleichverteilung. Problemstellungen mit J=10 Aktivitäten. (Für zwei Probleme werden nur in den VNS-Algorithmen zulässige Lösungen gefunden.)

	Bessere Lösung gefunden			
	deterministisch	gleichwertig	stochastisch	p-Wert (*)
GA	5	1	22	.002
VNS	5	1	24	< .001
Beide Algorithmen	5	1	24	< .001

(*) $H_0: \pi = 0.5$ Gleichwertigkeit zwischen deterministisch/stochastisch, unentscheidbare Lösungen werden eliminiert

4 Diskussion

Die Ergebnisse zeigen, dass für die hier betrachteten Problemstellungen der VNS dem GA in den meisten Fällen überlegen ist. Eine mögliche Begründung liegt hier in der betrachteten Anzahl an Lösungen. Der genetische Algorithmus verwendet zwar eine bestimmte Populationsgröße sowie eine vorgegebene Anzahl an Generationen. Dennoch besteht eine Population innerhalb einer Generation aus zwei Gründen nicht aus lauter verschiedenen Lösungen. Erstens ergibt es sich durch den Selektionsmechanismus, dass manche Lösungen (vornehmlich die besseren) mehrmals vertreten sind (siehe auch Appendix, Abschnitt 6.4). Dies ist immerhin im Sinne des Algorithmus, da darauf die Idee aufbaut, dass sich über die Zeit die besseren Lösungen durchsetzen und so stetig eine Verbesserung stattfindet. Zweitens wird in der hier verwendeten Version des GA die Vorgängerzulässigkeit nicht mitberücksichtigt, um *crossover* und Mutation einfach zu gestalten. Somit können auch Lösungen auftreten, die zwar prinzipiell unterschiedlich sind, jedoch aufgrund der Vorgängerbeziehungen zwangsläufig zur gleichen Zielfunktion führen (siehe Abschnitt 1.1.8). Dies reduziert weiter den im GA betrachteten Bereich an Lösungen.

Für den Vergleich mit dem VNS ergibt sich dadurch das Problem, dass die Anzahl an tatsächlich betrachteten Individuen im GA nicht im Vorhinein bestimmbar ist. Der VNS geht außerdem sparsamer mit den Ressourcen um, indem lediglich Vorgänger-zulässige Lösungen und noch nicht vorher berechnete Lösungen neu evaluiert werden. Damit ist die Gesamtanzahl an unterschiedlichen Lösungen relativ genau planbar (diese fließt als Parameter in den Algorithmus ein).

Obwohl die Maximalanzahl im VNS so gewählt ist, dass sich ein relativ fairer Vergleich zwischen den beiden Algorithmen ergibt, muss dies nicht für alle Problemstellungen gelten und kann somit dem VNS schon dadurch einen Vorteil verschaffen. Es muss allerdings beachtet werden, dass im VNS keine unnötigen Evaluierungen von Lösungen stattfinden, was unter dem Gesichtspunkt der Ressourceneffizienz natürlich wünschenswert ist.

Für den GA ließen sich diese Effizienz ebenfalls erreichen, was aber nur durch eine Steigerung der Komplexität des Algorithmus, besonders bei den Mechanismen *crossover* und Mutation, möglich ist. So müsste z.B. die Vorgänger-Zulässigkeit bei der Konstruktion von neuen Lösungen eingebunden werden. Eine andere Möglichkeit der Steuerung der Vielfalt in den Populationen könnte durch einen anderen Selektionsmechanismus als die *tournament selection* erreicht werden (obwohl der Selektionsdruck mit jeweils nur zwei gegeneinander antretenden Individuen für diese Art der Selektion so niedrig wie möglich gewählt ist).

Eine weitere Begründung kann auch in der Wahl der Parameter (Anteil an Elitismus; Wahrscheinlichkeit für *crossover* und Mutation) für den GA liegen. Eine differenziertere Betrachtung von mehreren Parameterkombinationen könnte hier eine Verbesserung der Performanz des Algorithmus bringen.

Bei der Bearbeitung der Problemstellungen wird der Einfluss der beiden Ressourcentypen (erneuerbar und nicht-erneuerbar) deutlich. Die erneuerbaren Ressourcen beeinflussen beim Erstellen der *schedules* direkt die *makespan* und damit die Zielfunktion. Die nicht-erneuerbaren hingegen beeinflussen prinzipiell nur die Zulässigkeit einer Lösung. Häufig wird das RCPSP ohne Berücksichtigung der nicht-erneuerbaren Ressourcen betrachtet. Die Berücksichtigung der nicht-erneuerbaren Ressourcen bringt allerdings einen Mehrwert, wenn man einen Praxisbezug für eine solche Art von Problemstellungen herstellen will. Beispiele für erneuerbare Ressourcen könnten Arbeitskraft, verfügbare technologische Ressourcen (z.B. Rechenzeit, Energie) oder räumliche Ressourcen (Lagerplatz) sein. Nicht erneuerbare Ressourcen können z.B. als Gesamtbudget oder ein Gesamtkontingent an Arbeitsstunden in einem realen Problem auftreten. Bei der Optimierung entstehen durch die Berücksichtigung der nicht-erneuerbaren Ressourcen allerdings einige Hürden, die die Komplexität der Probleme und der Algorithmen erhöhen. In dieser Arbeit wird die Zielfunktion bei einer Überschreitung der Ressourcen mit einem Strafterm erhöht. Dabei wird mit fortschreitendem Verlauf der Algorithmen das Ausmaß der Pönalisierung erhöht, sodass gegen Ende der Laufzeit de facto nur mehr zulässige Lösungen als beste Lösung in Frage kommen. Dieser Ansatz könnte allerdings noch verfeinert werden. Eine Möglichkeit wäre, das Problem als multikriterielles Optimierungsproblem aufzufassen, wobei sowohl die *makespan* (bzw. der $AVaR_\alpha$) als auch das Ausmaß der Überschreitungen der einzelnen Ressourcen minimiert werden.

Die Betrachtung der stochastischen Bearbeitungszeiten, sowie die Wahl des $AVaR_\alpha$ als Zielfunktion erhöhen den Aufwand bei der Berechnung der Zielfunktion beträchtlich. Die Vermeidung der Berechnung von unnötigen Lösungen ist daher in den Optimierungsalgorithmen angebracht. Beim VNS lässt sich dieser Aspekt leichter umsetzen, da hier die Nachbarschaften durch Herstellen von Vorgänger-Zulässigkeit der Lösungen beträchtlich reduziert werden können. Einiges Einsparpotential geht allerdings gerade durch die Berechnung der Zielfunktion auf Basis von Szenarien verloren. In deterministischen Problemstellungen kann eine Lösung bereits verworfen werden, wenn beim *scheduling* die momentane *makespan* die gesamte *makespan* der aktuell besten Lösung überschreitet. Bei den hier betrachteten Problemstellungen ist das nicht möglich, da die Zielfunktion erst feststeht, wenn die *schedules* für alle Szenarien erstellt wurden, und daraus der $AVaR_\alpha$ berechnet wurde.

Dieser zusätzlichen Komplexität durch die Betrachtung von stochastischen Bearbeitungszeiten steht allerdings der höhere praktische Wert von solchen Betrachtungsweisen gegenüber. Selten werden praktische Probleme exakt bestimmbare Bearbeitungszeiten der Aktivitäten aufweisen, sodass die Modellierung als Zufallsvariable sicherlich häufig angebracht ist.

Bei der Betrachtung des *value of the stochastic solution* zeigt sich, dass bei den ursprünglich betrachteten Problemstellungen die stochastischen Algorithmen nicht so sehr im Vorteil sind wie erwartet. Ein möglicher Grund ist die Art der Konstruktion der Zufallsvariablen. Die Modellierungen sind so gewählt, dass die Varianz eine monoton wachsende Funktion der Basiszeit B darstellt. Nun entspricht aber auch jeweils der Modalwert bzw. (annähernd der Erwartungswert) der Basiszeit. Somit ist in den hier betrachteten Modellen aber auch der $AVaR_\alpha$ der Zufallsvariablen proportional zu den Basiszeiten. Dies fördert letztendlich einen starken Zusammenhang zwischen der *makespan* auf Basis der Basiszeiten und dem $AVaR_\alpha$. Wird die Konstruktion anders aufgebaut, d.h. mit relativ hoher Varianz bei geringen Basiszeiten und relativ geringer Varianz bei hohen Basiszeiten, tritt der Vorteil der stochastischen Behandlung der Problemstellungen deutlicher zutage, d.h. die unter Berücksichtigung der stochastischen Zielfunktion gefundenen Lösungen sind meistens besser, als die auf Basis der *makespan* auf Basis der deterministischen Basiszeiten gefundenen Lösungen.

Zusammenfassend lässt sich sagen, dass beim Vergleich der Algorithmen der VNS etwas besser abschneidet. Die Implementierung ist leichter und die Lösungsgüte häufig besser. Die Umsetzung des GA weißt allerdings noch Verbesserungspotenzial auf. Die hier betrachteten Problemstellungen erweisen sich als recht komplex (stochastische Bearbeitungszeiten, Einbezug aller möglichen Ressourcen), was sich auf die erforderliche Komplexität der Algorithmen niederschlägt. Dennoch sollten aufgrund des erhöhten praktischen Bezugs diese Informationen weiterhin bei diesen Problemstellungen nicht vernachlässigt werden.

5 Literatur

Alcaraz, J., Maroto, C., & Ruiz, R. (2003). Solving the Multi-Mode Resource-Constrained Project Scheduling Problem with genetic algorithms. *Journal of Operational Research Society, 54*, 614-626.

Ballestín, F. (2007). When it is worthwhile working with the stochastic RCPSP? *Journal of Scheduling, 10*, 153-166.

Ballestín, F., & Leus, R. (2009). Resource-constrained project scheduling for timely project completion with stochastic activity durations. *Production and Operations Management, 18*, 459-474.

Birge, J. R., & Louveaux, F. (2010). *Introduction to stochastic programming* (2nd Edition). New York: Springer.

Blasewicz, J., Lenstra, J. K., & Rinnoy Kan, A. H. G. (1983). Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics, 5*, 11-24.

Bruckner, P., Drexl, A., Möhring, R., Neumann, K., & Pesch, E. (1999). Resource-constrained project scheduling: notation, classification, models, and methods. *European Journal of Operational Research, 112*, 3-41.

Falkenauer, E., & Bouffouix, S. (1991). A genetic algorithm for job shop. In *Proceedings of the 1991 IEEE international conference on robotics and automation* (Vol.1), 824-829.

Forbes, C., Evans, M., Hastings, N., & Peacock, B. (2010). *Statistical Distributions* (4th Edition). Hoboken: John Wiley & Sons.

Gutjahr, W. (2015). Bi-objective multi-mode project scheduling under risk aversion. *European Journal of Operational Research, 246*, 421-434.

Hansen, P., Mladenović, N., & Moreno Pérez, J. A. (2010). Variable neighbourhood search: methods and applications. *Annals of Operational Research, 175*, 367-407.

Hartmann, S. (2001). Project scheduling with multiple modes: a genetic algorithm. *Annals of Operations Research, 102*, 111-135.

Hartmann, S., & Briskorn, D. (2010). A survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research, 207*, 1-14.

Holland, J. H. (1975). *Adaption in natural and artificial systems*. Ann Arbor: University of Michigan Press.

Kolisch, R., & Drexl, A. (1997). Local search for nonpreemptive multi-mode resource-constrained project scheduling. *IIE Transactions, 29*, 987-999.

Kolisch, R., & Hartmann, S. (1999). Heuristic algorithms for the resource-constrained project scheduling problem: classification and computational analysis. In *Project scheduling*. 147-178. New York: Springer.

Kolisch, R., & Padman, R. (2001). An integrated survey of deterministic project scheduling. *Omega, 29*, 249-272.

Kolisch, R., & Sprecher, A. (1996). PSPLIB – a project scheduling problem library: OR software-ORSEP operations research software exchange program. *European Journal of Operational Research, 96*, 205-216.

Liaw, C.-F. (2000). A hybrid genetic algorithm for the open shop scheduling problem. *European Journal of Operational Research, 124*, 28-42.

Özdamar, L. (1999). A genetic algorithm approach to a general category project scheduling problem. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), 29*, 44-59.

Özdamar, L., & Gunduz, U. (1995). A survey on the resource-constrained project scheduling problem. *IIE Transactions, 27*, 574-586.

R Core Team. (2017). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL: https://www.R-project.org/.

Razali, N. M., & Geraghty, J. (2011). Genetic algorithm performance with different selection strategies in solving TSP. *Proceedings of the world congress on engineering* (Vol.2), 1134-1139.

Sivanandam, S. N., & Deepa, S. N. (2008). *Introduction to genetic algorithms.* Berlin Heidelberg: Springer.

Umbarkar, A. J., & Sheth, P. D. (2015). Crossover operators in genetic algorithms: a review. *ICTACT Journal on Soft Computing, 6*, 1083-1092.

Whitley, D. (1994). A genetic algorithm tutorial. *Statistics and Computing, 4*, 65-85.

6 Appendix

6.1 Zusammenfassung

Ziel der Arbeit ist es, zwei heuristische Optimierungsmethoden für das multi-modale *Resource-Constrained Project Scheduling Problem* (RCPSP) zu erstellen und zu evaluieren. Dabei werden die Bearbeitungszeiten der Aktivitäten stochastisch (d.h. als Zufallsvariablen) angenommen und drei Modellierungen konstruiert. Als Basismodelle werden Gleichverteilung, Dreiecksverteilung und eine verschobene Weibull-Verteilung herangezogen.

Als Heuristiken werden dabei die *variable neighborhood search* (VNS) sowie ein genetischer Algorithmus (GA) verwendet. Diese Algorithmen müssen auf die besondere Problemstruktur, die stochastischen Bearbeitungszeiten sowie die in dieser Arbeit verwendete spezielle Zielfunktion – dem *average value at risk* zum Niveau α ($AVaR_\alpha$) der *makespan* maßgeschneidert konstruiert werden. Die Evaluierung erfolgt für insgesamt 90 Projekte der Benchmark-Datenbank PSPLIB. Die Projekte sind dabei von unterschiedlicher Größenordnung und Schwierigkeit. Aufgrund der stochastischen Modellierung der Bearbeitungszeiten ist eine Erweiterung der aus der Datenbank entnommenen Projekte durch die Konstruktion von Szenarien auf Basis der Modellierungen der stochastischen Bearbeitungszeiten nötig.

Der Vergleich der Algorithmen zeigt, dass der VNS-Algorithmus in der hier verwendeten Umsetzung häufig besser abschneidet als der GA. Es werden mit der VNS im Großteil der betrachteten Problemstellungen bessere Lösungen gefunden. Die Komplexität der Problemstellung und der Modellierung birgt einige Hürden, die bei der Umsetzung der Algorithmen beachtet werden müssen. Insofern ergibt sich noch Verbesserungspotenzial um die Performanz der Algorithmen zu steigern.

Die stochastische Modellierung der Bearbeitungszeiten ist zwar praxisrelevant, erfordert jedoch eine höhere Komplexität der Algorithmen, beispielsweise durch die aufwändigere Berechnung der Zielfunktion. Im Vergleich mit einer deterministischen Variante der Algorithmen erweisen sich die stochastischen Modellierungen jedoch als vorteilhaft. Dieser Vorteil zeigt sich insbesondere dann, wenn Zufallsmodelle zu Grunde liegen, in denen der Zusammenhang zwischen $AVaR_\alpha$ und Modalwert (bzw. Erwartungswert) nicht mehr stark ausgeprägt ist.

6.2 Evaluierte Problemstellungen

Tabelle 14: Analysierte Probleminstanzen der PSPLIB

J10	**J16**	**J30**
j1010_7	j1611_8	j3014_3
j1011_3	j1612_2	j3015_10
j1019_6	j1614_4	j3015_2
j1022_6	j1622_2	j3015_4
j1026_5	j1624_2	j3015_7
j1026_6	j1626_8	j3017_4
j1028_1	j1627_8	j3020_5
j1029_10	j1630_10	j3020_6
j1029_7	j1632_10	j3024_5
j1030_1	j1634_4	j3026_9
j1031_1	j1635_2	j3027_1
j1035_4	j1635_3	j3029_5
j1038_2	j1637_1	j303_10
j1039_1	j1638_6	j303_6
j1039_5	j1640_8	j3031_7
j1040_5	j1641_1	j3034_3
j1044_3	j1641_10	j3036_10
j1046_4	j1642_4	j3038_2
j1047_9	j1646_2	j304_7
j1052_9	j1647_4	j3040_7
j1054_2	j1650_4	j3044_9
j1055_6	j1650_9	j3048_8
j1055_8	j1656_10	j3051_6
j1056_6	j1656_5	j3052_6
j1056_8	j1656_6	j3052_7
j1058_10	j1657_1	j3053_5
j1058_7	j1658_2	j3059_3
j1060_2	j1659_1	j3059_8
j1062_1	j166_2	j306_7
j1064_6	j1663_6	j308_6

6.3 Neue Modellierung der Bearbeitungszeiten

Als neue Modellierung wird dabei eine Gleichverteilung um die Basiszeiten B verwendet. Dabei steigt jedoch die Varianz nicht mehr generell linear in B an, sondern sinkt für höhere Basiszeiten wieder ab. Formal lässt sich die Modellierung folgendermaßen beschreiben:

$$X \sim U(a, b)$$

mit

$$a = \left(B - \frac{3}{B}\right) \cdot \mathbb{1}_{\{B \geq 6\}}$$

und

$$b = 2 \cdot B \cdot \mathbb{1}_{\{B < 6\}} + \left(B + \frac{3}{B}\right) \cdot \mathbb{1}_{\{B \geq 6\}}$$

Abbildung 15 zeigt die Dichten der so konstruierten Zufallsvariablen für einige Beispiele von Basiszeiten. Die so konstruierten Zufallsvariablen sind stark auf die in den aus der PSPLIB entnommenen Problemstellungen maßgeschneidert. Die dort vorkommenden Basiszeiten sind ganzzahlig und liegen im Bereich zwischen 1 und 10. Die Verteilungen sind in jedem Fall symmetrisch um B, sodass die deterministischen Lösungen der ursprünglichen Gleichverteilungen verwendet werden können. Prinzipiell ist hier jedoch die Idee, dass für höhere Bearbeitungszeiten die Varianz gering ist. Es treten dabei auch Fälle auf, bei denen der $AVaR_\alpha$ (für das hier verwendete $\alpha = 0.2$) niedriger ist, obwohl der Modalwert bzw. Erwartungswert B höher ist.

Beispielsweise ist für eine Basiszeit $B = 5$ der Modalwert gleich der Basiszeit, d.h. $Mod(X) = 5$. Der theoretische $AVaR_\alpha$ beträgt in diesem Fall $AVaR_{0.2}(X) = 9$. Für die Basiszeit $B = 6$ beträgt der Modalwert wiederum $Mod(X') = 6$. Nun ist nach der obigen Modellierung jedoch $AVaR_{0.2}(X) = 6.4$.

Die Annahme ist nun, dass in der Optimierung dieser Effekt sich auch auf den $AVaR_\alpha$ der *makespan*, und somit auf die Zielfunktion überträgt. Demnach sollte mit dieser Modellierung die Überlegenheit der stochastischen Algorithmen deutlicher zutage treten, da diese eben auf diese stochastische Zielgröße abzielen, im Gegensatz zu den deterministischen Algorithmen. Diese verwenden klarerweise wieder den Modalwert, d.h. die Basiszeiten B und optimieren die einfache *makespan*.

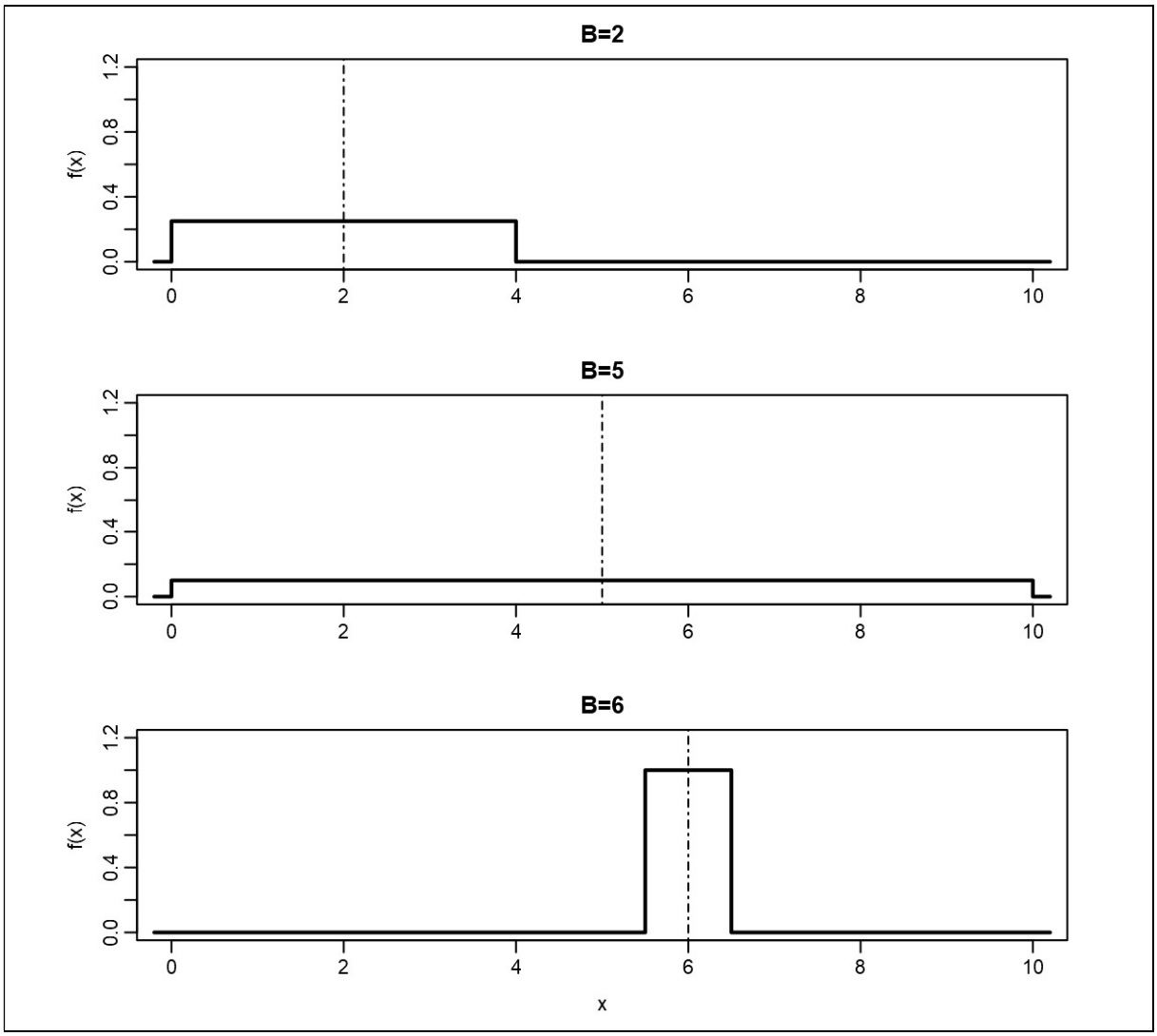

Abbildung 15: Dichtefunktionen der Gleichverteilungen nach der neuen Konstruktion für unterschiedliche Basiszeiten B.

6.4 Diversität der Populationen im GA

Der Selektionsmechanismus beim GA bewirkt, dass in jeder Generation ein bestimmter Anteil an Individuen mehrfach vorkommt. D.h. eine Population enthält nur einen bestimmten Anteil an unterscheidbaren Individuen. Interessant ist nun, wie sich dieser Anteil über die Zeit entwickelt. Abbildung 16 gibt für alle 270 Durchgänge die Anzahl an unterscheidbaren Individuen für jede der 1000 Generationen. Zu Beginn sinkt die Anzahl meist relativ schnell ab. Innerhalb der ersten 50 Iterationen pendelt sich der Wert dann aber meisten Durchgängen im Bereich um ca. 60 % einpendelt (Populationsgröße = 100). Trotzdem gibt es einige Durchgänge, bei denen die Populationsgröße auch während dem Verlauf teilweise sehr stark absinkt.

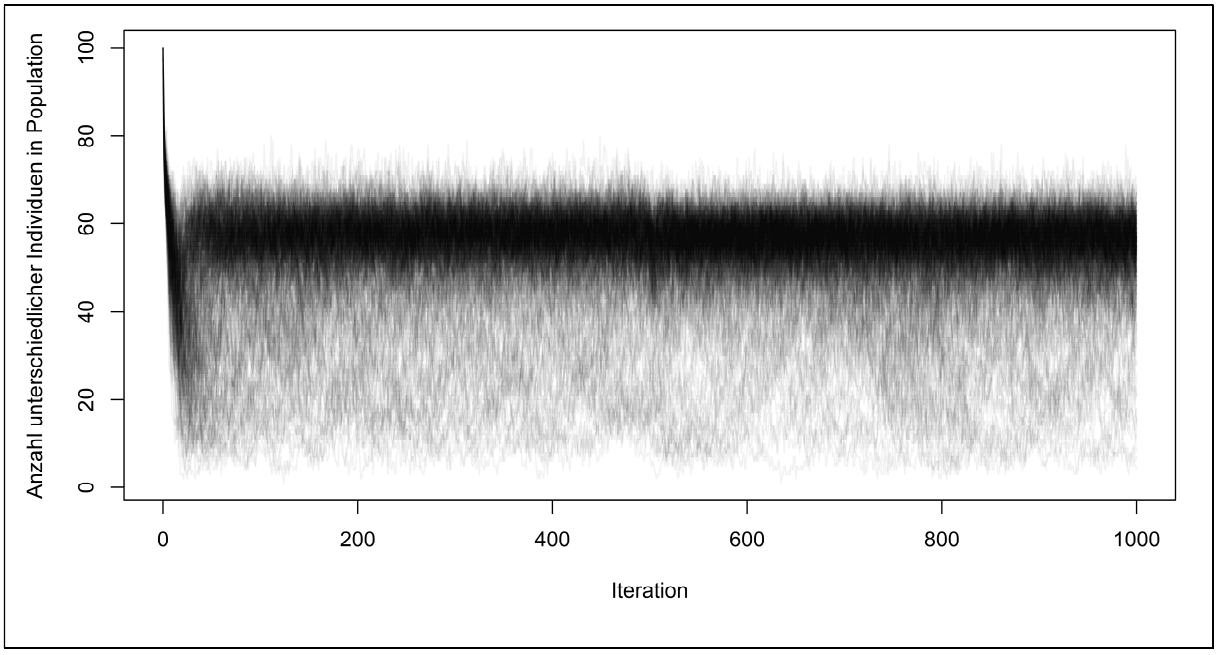

Abbildung 16: Anzahl unterscheidbarer Individuen pro Iteration für 270 Durchgänge des GA.

Interessant ist auch der leichte Knick um Iteration 500. Dies ist der Zeitpunkt, ab dem die unzulässigen Lösungen bestraft werden, d.h. ab hier werden vornehmlich zulässige Lösungen in der Population behalten, da sich die Bestrafung natürlich umgekehrt proportional auf die Fitness auswirkt. Abbildung 17 enthält den mittleren Verlauf, d.h. für jede Iteration den Mittelwert der Anzahl über alle 270 Durchgänge. Außerdem sind drei exemplarische Verläufe abgebildet. Diese sind der Übersichtlichkeit halber geglättet (laufender Mittelwert über 40 Iterationen).

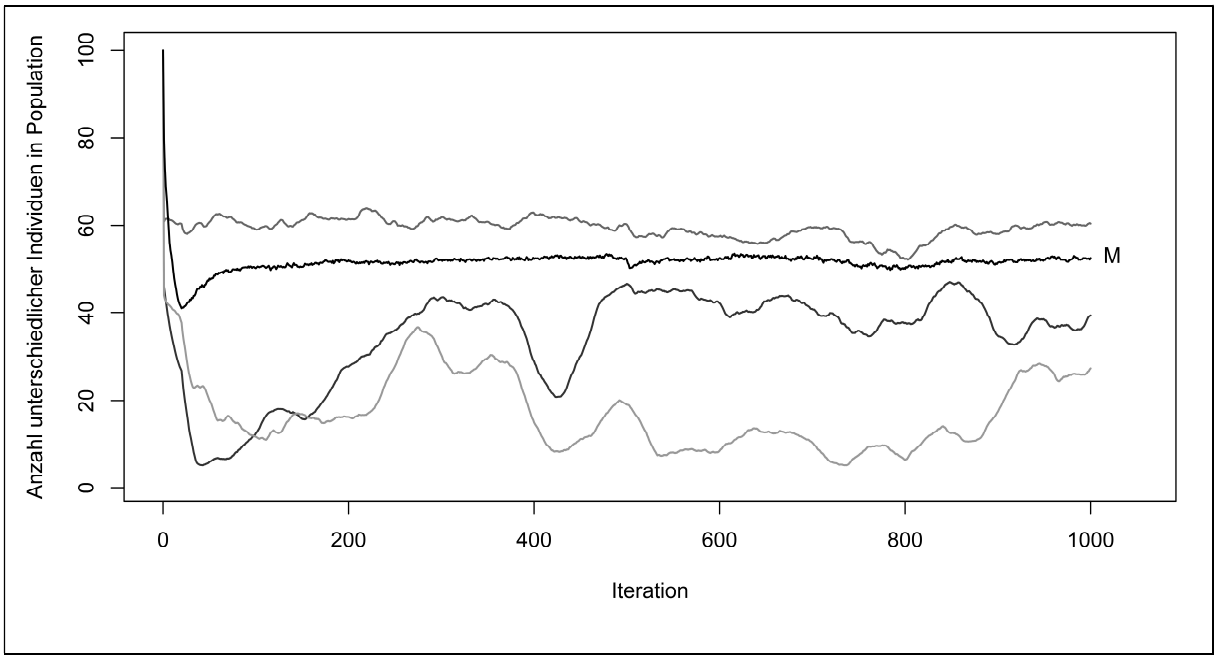

Abbildung 17: Mittlerer Verlauf (M) und drei exemplarische, geglättete Verläufe der Diversität der Populationen im GA.

Diese Beispiele zeigen, dass für manche Durchgänge durchwegs recht vielfältige Populationen evaluiert werden. Einige weisen eine eher niedrige Diversität über den Verlauf des Algorithmus auf. Interessant wäre natürlich, wodurch die Art des Verlaufs bestimmt wird, bzw. ob ein Einfluss auf die Lösungsgüte zu beobachten ist (dies wird in dieser Arbeit nicht behandelt).

6.5 R-Code für ausgewählte Funktionen

Dieser Abschnitt enthält zur Demonstration einige Auszüge des R-Codes, der zur Bearbeitung der Problemstellungen verwendet wurde. Dabei handelt es sich um die Funktionen zum Erstellung eines *schedules*, der Berechnung der Zielfunktion auf Basis von gegebenen Szenarien und Lösungen sowie den beiden Kernstücken der Arbeit – dem VNS-Algorithmus und dem GA. Es ist allerdings zu beachten, dass alle Funktionen von einer Vielzahl von weiteren Funktionen/Inputs abhängen, die hier aus Platzgründen nicht alle angegeben werden können.

6.5.1 Erstellung eines *schedules*

```
##############################################################################
# Scheduling a project
##############################################################################

sced_fun <- function(dur,
                     prio,
                     prec,
                     res_req,
                     res_cap){
    j_num <- length(dur)                              # number of jobs (including dummy jobs)
    completed <- rep(FALSE, times = j_num)            # binary coding of completed jobs
    m_span <- 0                                       # objective function
    act_jobs <- NULL                                  # active jobs
    act_res <- rep(0, times = length(res_cap))        # active resource request
    pr_vec <- prio
    dur_rest <- dur

    while(sum(completed) < j_num){
        if(length(pr_vec) > 1){
            j_ready <- pr_vec[apply(prec[pr_vec,], MARGIN = 1,
                                    FUN = function(x) all(completed[as.logical(x)]))]
        } else {
            j_ready <- pr_vec
        }

        #start jobs according to priority list
        for(i in seq_along(j_ready)){

            j <- j_ready[i]
            if(!any(res_cap-res_req[j,]-act_res < 0)){
                act_jobs <- c(act_jobs, j)
                act_res <- colSums(res_req[act_jobs,])
                pr_vec <- pr_vec[!(pr_vec == j)]
            }
        }
        # passing time until next decision point
        passed <- min(dur_rest[act_jobs])
        dur_rest[act_jobs] <- (dur_rest[act_jobs]-passed) * (dur_rest[act_jobs] - passed > 0)
        step_comp <- act_jobs[which(dur_rest[act_jobs] == 0)]
        completed[step_comp] <- T
        act_jobs <- act_jobs[-which(dur_rest[act_jobs] == 0)]
        act_res <- colSums(res_req[act_jobs,])
        if(length(act_jobs) == 0) act_res <- rep(0, times = length(res_cap))
        m_span <- m_span + passed
    }
    m_span
}
```

6.5.2 Berechnung der Zielfunktion

```
##############################################################################
# objective function and some other measures
##############################################################################

rcp_eval_trace_p <- function(sol,
                             project,
                             dist = "uniform",
                             alpha = 0.2,
                             nr_const = 1e+10,
                             paral = TRUE,
                             det = FALSE){
    # project: project, as built in list form by scen_sim
    # sol: solution in form of a list of one list of two vectors of the following form:
    # [[1]]
    # sol[[1]]: priority list (part of a policy)
    # sol[[2]]: vector of modes, to use for each of the jobs
    # dist: which of the three solutions should be used ("triangular", "uniform", "weibull")
    # alpha: value at risk is calculated at the (1-alpha)-quantile
    # nr_const: constant for weighing the exceeding nonrenewable resources
    #           (optional: set nr_const = c(x, 0) for a positive value x to only
    #           consider one renewable resource)
    # paral: should parallelization be used
    # det: should only the deterministic values be used

    if(dist == "uniform") dur_mat <-
            project$simulations.unif.simulations[mode_sel(project = project,
                                                          mod_list = sol[[2]]),]
    if(dist == "triangular") dur_mat <-
            project$simulations.triang.simulations[mode_sel(project = project,
                                                            mod_list = sol[[2]]),]
    if(dist == "weibull") dur_mat <-
            project$simulations.weibull.simulations[mode_sel(project = project,
                                                             mod_list = sol[[2]]),]

    if(det == TRUE){
        dur_mat <- matrix(matrix(project$durations,
                                 ncol = 1)[mode_sel(project = project,
                                                    mod_list = sol[[2]]),],
                          ncol = 1)
    }

    if(paral == TRUE){

        dur_list <- split(t(dur_mat), seq_len(ncol(dur_mat)))
        library(parallel)
        n_clust <- detectCores() - 1

        cl <- makeCluster(n_clust)
        clusterEvalQ(cl, library(mahro))
        clusterExport(cl, c("dur_list", "sol"), environment())
        span_vec <-
            unlist(parLapply(cl, dur_list, fun = sced_fun, prio = sol[[1]],
                             prec = project$precedence,
                             res_req = project$res.request[mode_sel(project = project,
                                                                    mod_list = sol[[2]]),
1:2],
                             res_cap = project$res.capacity[1:2]))
        stopCluster(cl)

    } else {

        span_vec <-
            apply(dur_mat, MARGIN = 2, FUN = sced_fun, prio = sol[[1]],
                  prec = project$precedence,
                  res_req = project$res.request[mode_sel(project = project,
                                                         mod_list = sol[[2]]), 1:2],
                  res_cap = project$res.capacity[1:2])
    }

    if(det != TRUE){
        nr_exc <- sapply(nr_exceed_fun(project = project, sol = sol),
                         FUN = function(x) max(0, x))
        AVaR_span <- av_val_risk(span_vec, alpha = alpha)

        c(AVaR_span = AVaR_span,
```

```
          obj_fun = AVaR_span + sum(nr_exc*nr_const),
          n1_exceed = nr_exc[1],
          n2_exceed = nr_exc[2],
          n1_const = nr_const[1],
          n2_const = ifelse(length(nr_const) > 1, nr_const[2], nr_const[1]),
          mean_span = mean(span_vec),
          min_span = min(span_vec),
          max_span = max(span_vec))

    } else {
        nr_exc <- sapply(nr_exceed_fun(project = project, sol = sol),
                         FUN = function(x) max(0, x))

        c(AVaR_span = as.vector(span_vec),
          obj_fun = as.vector(span_vec) + sum(nr_exc * nr_const),
          n1_exceed = nr_exc[1],
          n2_exceed = nr_exc[2],
          n1_const = nr_const[1],
          n2_const = ifelse(length(nr_const) > 1, nr_const[2], nr_const[1]),
          mean_span = mean(span_vec),
          min_span = min(span_vec),
          max_span = max(span_vec))
    }
}
```

6.5.3 VNS-Algorithmus

```
##############################################################################
# VNS-Algorithm
##############################################################################

vns_alg_new_trace <- function(project,
                              sol_max = 1e4,
                              alpha = 0.2,
                              k_max = 10,
                              max_try_at_kmax = 20,
                              dist = "uniform",
                              nr_const_fun = stand_nr_const_fun,
                              paral = c("pop", "proj", "none"),
                              lib.loc = NULL,
                              cluster_out = TRUE,
                              initial_sol = NULL,
                              stop_time = Inf,
                              det = FALSE){
    # project: a single project as produced by the listed result of the function scen_sim
    # sol_max: maximum number of total solutions considered
    # k_max: maximum width of the neighborhood (as used in the function neigh_sol)
    # alpha: parameter used for the average value at risk in rcp_eval
    # dist: parameter used for the choice of the distribution in rcp_eval
    # nr_const_fun: function to calculate the penalty for exceeding non-renewable
    #               resources
    # paral: parallelization mode
    # lib.loc: location of library mahro (used for parallelization)
    # cluster_out: used for parallelization --> builds cluster once at function
    #              level if TRUE
    # inital_solution: initial solution for the algorithm
    # stop_time: in addition to sol_max the processing can be stopped after this
    #            time point is reached
    # det: should only the deterministic values be used

    # initialize parameters etc.
    paral <- paral[1]

    # construction of clusters for parallelisation
    if(cluster_out == TRUE){
        library(parallel)
        n_clust <- detectCores() - 1
        cl <- makeCluster(n_clust)
        clusterExport(cl, "lib.loc", environment())
        clusterEvalQ(cl, library("mahro", lib.loc = lib.loc))

    } else {
        cl <- NULL
    }

    # initial values and useful variables
    n_jobs <- dim(project$precedence)[1]
    n_modes <- project$modes
    iter_nr <- 1
    iter_max_nr <- 1000

    iter <- 1
    k <- 0
    try_at_kmax <- 0
    updated_sol <- FALSE

    nr_const <- nr_const_fun(iter = iter_nr, iter_max = iter_max_nr)
    if(length(nr_const) == 1) nr_const <- rep(nr_const, times = 2)
    if(length(nr_const) > 2) nr_const <- nr_const[1:2]

    nr_const_new <- nr_const_old <- nr_const

    # initialize known data.frame and selection vector
    known_df <- as.data.frame(matrix(NA,
                                     nrow = 2 * sol_max,
                                     ncol = 11,
                                     dimnames = list(seq_len(2 * sol_max),
                                                     c("ID",
                                                       "feasible",
                                                       "AVaR_span",
                                                       "obj_fun",
                                                       "n1_exceed",
                                                       "n2_exceed",
```

```
                                                    "n1_const",
                                                    "n2_const",
                                                    "mean_span",
                                                    "min_span",
                                                    "max_span")
                                    )))
    known_df[["ID"]] <- as.character(known_df[["ID"]])

    known_sel <- rep(FALSE, times = 2 * sol_max)

    stop_eval <- FALSE
    stop_reason <- NULL

    # transformed precedence matrix for usage in later loop
    tr_prec_mat <- precedence_transform(project[["precedence"]],
                                        logical = T)

    # trace
    res_trace <- list(
        best_solutions = as.data.frame(
            matrix(NA, nrow = 2 * sol_max, ncol = 11,
                   dimnames = list(NULL,
                                   c("ID",
                                     "feasible",
                                     "AVaR_span",
                                     "obj_fun",
                                     "n1_exceed",
                                     "n2_exceed",
                                     "n1_const",
                                     "n2_const",
                                     "mean_span",
                                     "min_span",
                                     "max_span")))),
        best_feas_solutions = as.data.frame(
            matrix(NA,  nrow = 2 * sol_max, ncol = 5,
                   dimnames = list(NULL,
                                   c("ID",
                                     "AVaR_span",
                                     "mean_span",
                                     "min_span",
                                     "max_span")))),

        new_solutions = numeric(2 * sol_max),
        step_id = character(2 * sol_max),
        time_stamp = rep(Sys.time(), times = 2 * sol_max))

    # initial solution (has to be feasible; at least try often to construct feasible intial
sol)
    if(is.null(initial_sol)){

        popul_feas_test <- FALSE
        sec_iter <- 1

        while(!popul_feas_test & sec_iter < 10000){
            popul <- lapply(rep(n_jobs, times = 100),
                            FUN = function(x){
                                list(sol_prio = sample(seq_len(n_jobs),
                                                       size = n_jobs),
                                     sol_mode = sapply(
                                         n_modes,
                                         FUN = function(x) sample(seq_len(x),
                                                                  size = 1)))
                            })

            popul_feas <- !sapply(popul, FUN = function(x, y){
                any(nr_exceed_fun(project = y, sol = x) > 0)},
                y = project)

            popul_feas_test <- any(popul_feas)

            sec_iter <- sec_iter + 1
        }

        if(popul_feas_test){
            initial_sol <- popul[popul_feas][[1]]
        } else {
            initial_sol <- popul[[1]]
        }
    }
```

```
initial_sol <- precedence_validate_sol(initial_sol, project = project)

# evaluate initial solution
res_initial <- rcp_eval_trace_p(sol = initial_sol, paral = FALSE,
                                project = project, alpha = alpha,
                                dist = dist, nr_const = nr_const,
                                det = det)

# update known cases with initial solution
known_sel[1] <- TRUE
known_df[1, "ID"] <- sol_to_ID(initial_sol)
known_df[1, "feasible"] <- all(res_initial[c("n1_exceed", "n2_exceed")] == 0)
known_df[1, -(1:2)] <- res_initial

n_known <- 1

# update best solution with initial solution
best_sol <- list(obj_fun = res_initial["obj_fun"],
                 sol = initial_sol,
                 res_eval = known_df[1,])

# initial solution to trace
res_trace[["best_solutions"]][iter,] <- best_sol[["res_eval"]]

if(known_df[1, "feasible"]){
    res_trace[["best_feas_solutions"]][iter, ] <- known_df[1, c("ID",
                                                                "AVaR_span",
                                                                "mean_span",
                                                                "min_span",
                                                                "max_span")]
}
res_trace[["new_solutions"]][iter] <- 1
res_trace[["step_id"]][iter] <- "initial"
res_trace[["time_stamp"]][iter + 1] <- Sys.time()

# set test_sol to initial solution
test_sol <- initial_sol


# vns-loop
while(!stop_eval){

    iter <- iter + 1
    iter_nr <- ceiling(iter_max_nr * n_known / sol_max)
    nr_const_old <- nr_const_new
    nr_const_new <- rep(nr_const_fun(iter_nr, iter_max_nr), t = 2)

    # shake solution if necessary
    if(k > 0){
        test_sol <- precedence_validate_sol(
            sol = shake_move_random(sol = best_sol[["sol"]],
                                    n_modes = project[["modes"]],
                                    k = k),
            tr_prec_mat = tr_prec_mat)
    }

    # produce local neighborhood around test-solution
    neigh <- local_neighborhood(sol = test_sol, project = project,
                                valid = TRUE, tr_prec_mat = tr_prec_mat)
    neigh_ID_vec <- sapply(neigh, FUN = sol_to_ID)
    unknown_sel <- !(neigh_ID_vec %in% known_df[known_sel, "ID"])

    # update known cases (if necessary, i.e. if nr_const has changed)
    if(!all(nr_const_old == nr_const_new) & any(!known_df[["feasible"]],
                                                na.rm = T)){
        known_df[known_sel & !known_df[["feasible"]],] <-
            update_known_solutions(known_df[known_sel & !known_df[["feasible"]],],
                                   nr_const_new = nr_const_new)
    }

    # evaluate unknown cases
    if(any(unknown_sel)){
        neigh_res_new <- pop_eval_tr(pop = neigh[unknown_sel], paral = paral,
                                     project = project,
                                     alpha = alpha, dist = dist,
                                     nr_const = nr_const_new,
                                     lib.loc = lib.loc, cl = cl,
                                     det = det)
```

```
        known_df[seq_len(sum(unknown_sel)) + n_known,] <-
            data.frame("ID" = neigh_ID_vec[unknown_sel],
                       "feasible" = colSums(neigh_res_new[c("n1_exceed",
                                                            "n2_exceed"),]) == 0,
                       t(neigh_res_new),
                       stringsAsFactors = FALSE)
        n_known <- n_known + sum(unknown_sel)
        known_sel[seq_len(n_known)] <- TRUE
    }

    # update best solution
    new_best <- min(known_df[known_sel, "obj_fun"])

    if(new_best < best_sol[["obj_fun"]]){
        best_sol[["obj_fun"]] <- new_best
        best_sol[["res_eval"]] <- known_df[order(known_df[,"obj_fun"])[1],]
        best_sol[["sol"]] <- ID_to_sol(best_sol[["res_eval"]][["ID"]])

        updated_sol <- TRUE
    }

    # trace
    res_trace[["best_solutions"]][iter,] <- best_sol[["res_eval"]]

    if(all(best_sol[["res_eval"]][c("n1_exceed", "n2_exceed")] == 0)){
        res_trace[["best_feas_solutions"]][iter, ] <-
            best_sol[["res_eval"]][c("ID","AVaR_span","mean_span",
                                     "min_span","max_span")]

    } else if(any(known_df[,"feasible"], na.rm = TRUE)){
        feas_df <- known_df[known_df[known_sel, "feasible"],
                            c("ID","AVaR_span","mean_span",
                              "min_span","max_span")]
        res_trace[["best_feas_solutions"]][iter, ] <-
            feas_df[order(feas_df[,"AVaR_span"]),][1,]
    }

    res_trace[["new_solutions"]][iter] <- sum(unknown_sel)
    res_trace[["step_id"]][iter] <- ifelse(k > 0, "shake", "descent")
    res_trace[["time_stamp"]][iter + 1] <- Sys.time()

    # update k if necessary
    if(updated_sol){
        k <- 0
        try_at_kmax <- 0
        updated_sol <- FALSE
    } else if(k < k_max){
        k <- k + 1
    } else {
        try_at_kmax <- try_at_kmax + 1
    }

    # update check of stopping criteria
    if(Sys.time() > stop_time){
        stop_reason <- c(stop_reason, "time_limit")
        stop_eval <- TRUE
    }

    if(n_known >= sol_max){
        stop_reason <- c(stop_reason, "maximal_solutions")
        stop_eval <- TRUE
    }

    if(try_at_kmax > max_try_at_kmax){
        stop_reason <- c(stop_reason, "maximum_try_at_kmax")
        stop_eval <- TRUE
    }

}
# end of function --> result and close cluster

# close clusters if necessary
if(cluster_out == TRUE) stopCluster(cl)

# best feasible solution so far
if(all(best_sol[["res_eval"]][c("n1_exceed", "n2_exceed")] == 0)){
    best_feasible_sol <- best_sol
    best_feasible_sol[["res_eval"]] <-
```

```
        best_feasible_sol[["res_eval"]][c("ID","AVaR_span","mean_span",
                                          "min_span","max_span")]
    } else if(any(known_df[,"feasible"], na.rm = TRUE)){
        feas_df <- known_df[known_df[known_sel, "feasible"],
                            c("ID","AVaR_span","mean_span",
                              "min_span","max_span")]

        best_feasible_sol <- list(sol = NULL,
                                  obj_fun = NULL,
                                  res_eval = feas_df[order(feas_df[,"AVaR_span"]),][1,])
        best_feasible_sol[["sol"]] <- ID_to_sol(best_feasible_sol[["res_eval"]][["ID"]])
        best_feasible_sol[["obj_fun"]] <- best_feasible_sol[["res_eval"]]["AVaR_span"]
    } else {
        best_feasible_sol <- NULL
    }

    # clean trace/results from initially too large columns/entries
    res_trace[["best_solutions"]] <- cbind(
        iter = seq_len(iter),
        res_trace[["best_solutions"]][seq_len(iter),])

    res_trace[["best_feas_solutions"]] <- cbind(
        iter = seq_len(iter),
        res_trace[["best_feas_solutions"]][seq_len(iter),])

    res_trace[["new_solutions"]] <- res_trace[["new_solutions"]][seq_len(iter)]

    res_trace[["step_id"]] <-     res_trace[["step_id"]][seq_len(iter)]

    res_trace[["time_stamp"]][iter + 2] <- Sys.time()
    res_trace[["time_stamp"]] <- res_trace[["time_stamp"]][seq_len(iter + 2)]

    # result
    list(best_feasible_sol = best_feasible_sol,
         trace = c(res_trace, list(known_df = known_df[known_sel,])),
         parameters = list(project = project,
                           sol_max = sol_max,
                           alpha = alpha,
                           k_max = k_max,
                           max_try_at_kmax = max_try_at_kmax,
                           dist = dist,
                           nr_const_fun = nr_const_fun,
                           paral = paral,
                           lib.loc = lib.loc,
                           cluster_out = cluster_out,
                           initial_sol = initial_sol,
                           stop_time = stop_time),
         stop_reason = stop_reason)
}
```

6.5.4 Genetischer Algorithmus

```
################################################################################
# Genetic algorithm
################################################################################

gen_alg_new_trace <- function(project,
                              iter_max = 1000,
                              n_pop = 100,
                              p_elite = 0.02,
                              alpha = 0.2,
                              dist = "uniform",
                              p_cross_mode = 0.2,
                              p_cross_prio = 0.2,
                              p_muta_mode = 0.02,
                              p_muta_prio = 0.02,
                              nr_const_fun = stand_nr_const_fun,
                              paral = c("pop", "proj", "none"),
                              lib.loc = NULL,
                              cluster_out = TRUE,
                              det = FALSE){
    # project: a single project as produced by the listed result of the function scen_sim
    # iter_max: maximum number of iterations (the number of generations considered
    #           in the algorithm)
    # n_pop: size of the population (of solutions) within each generation
    # alpha: parameter used for the average value at risk in rcp_eval
    # dist: parameter used for the choice of the distribution in rcp_eval
    # p_elite: upper proportion of the population which is passed from one generation
    #          without further evaluation;
    #          "elite" of the current population (best p*n individuals are passed
    #          to next generation)
    # p_cross_mode/p_cross_prio: probabilities for the occurrence of a cross_over
    #                            in mode_vec or prio_list
    # p_muta_mode/p_muta_prio: probabitlities for the occurrence of a mutation
    #                          in mode_vec or prio_list
    # nr_const_fun: function to calculate the penalty for exceeding non-renewable
    #               resources
    # paral: parallelization mode
    # lib.loc: location of library mahro (used for parallelization)
    # cluster_out: used for parallelization --> builds cluster once at function
    #              level if TRUE
    # det: should only the deterministic values be used

    # initial values and useful variables
    n_jobs <- dim(project$precedence)[1]
    n_modes <- project$modes
    iter <- 1
    nr_const <- nr_const_fun(iter = iter, iter_max = iter_max)
    if(length(nr_const) == 1) nr_const <- rep(nr_const, times = 2)
    if(length(nr_const) > 2) nr_const <- nr_const[1:2]

    n_elite <- ceiling(p_elite * n_pop)
    n_new <- n_pop - n_elite
    unknown <- rep(FALSE, times = n_pop)

    res_trace <- list(best_res = matrix(0, ncol = iter_max + 1, nrow = 9,
                                        dimnames = list(c("AVaR_span",
                                                          "obj_fun",
                                                          "n1_exceed",
                                                          "n2_exceed",
                                                          "n1_const",
                                                          "n2_const",
                                                          "mean_span",
                                                          "min_span",
                                                          "max_span"),
                                                        NULL)),
                      solutions = vector(mode = "list", length = iter_max + 1),
                      best_feas_res = matrix(NA, ncol = iter_max + 1, nrow = 4,
                                             dimnames = list(c("AVaR_span",
                                                               "mean_span",
                                                               "min_span",
                                                               "max_span"),
                                                             NULL)),
                      feas_solutions = vector(mode = "list", length = iter_max + 1),
                      diversity = numeric(length = iter_max + 1),
                      num_feasible = numeric(length = iter_max + 1),
                      full_eval_df = data.frame(),
                      names = c("start", seq_len(iter_max)),
```

```
                  time_stamp = rep(Sys.time(), times = iter_max + 2))

# number of cases selected for cross_over --> for convenience  must be multiples of two

n_cross_mode <- p_cross_mode * n_pop - (p_cross_mode * n_pop) %% 2
n_cross_prio <- p_cross_prio * n_pop - (p_cross_prio * n_pop) %% 2

cross_sel_mode <- c(rep(TRUE, times = n_cross_mode),
                    rep(FALSE, times = n_new-n_cross_mode))
cross_sel_prio <- c(rep(TRUE, times = n_cross_prio),
                    rep(FALSE, times = n_new-n_cross_prio))

# construction of clusters for parallelisation
if(cluster_out == TRUE){
    library(parallel)
    n_clust <- detectCores() - 1
    cl <- makeCluster(n_clust)
    clusterExport(cl, "lib.loc", environment())
    clusterEvalQ(cl, library("mahro", lib.loc = lib.loc))

} else {
    cl <- NULL
}

# random simulation of a starting population
# try to include at least one feasible solution
popul_feas_test <- TRUE
sec_iter <- 1

while(popul_feas_test & sec_iter < 10000){
    popul <- lapply(rep(n_jobs, times = n_pop),
                    FUN = function(x){
                        list(sol_prio = sample(seq_len(n_jobs),
                                               size = n_jobs),
                             sol_mode = sapply(
                                 n_modes,
                                 FUN = function(x) sample(seq_len(x), size = 1)))
                    })

    popul_feas_test <- any(!sapply(popul, FUN = function(x, y){
        any(nr_exceed_fun(project = y, sol = x) > 0)},
        y = project))

    sec_iter <- sec_iter + 1
}

# evaluation of the starting population and order + inital best solution
res_eval <- pop_eval_tr(pop = popul, paral = paral, project = project,
                        alpha = alpha, dist = dist, nr_const = nr_const,
                        lib.loc = lib.loc, cl = cl, det = det)
obj_vec <- res_eval["obj_fun", ]
ord_vec <- order(obj_vec)
ord_res <- res_eval[, ord_vec]
ord_pop <- popul[ord_vec]

full_df <- data.frame(ID = sapply(popul, FUN = sol_to_ID),
                      t(res_eval))


# which is best feasible solution
sel_feas_vec <- which(colSums(ord_res[c("n1_exceed", "n2_exceed"),]) == 0)
if(length(sel_feas_vec) > 0){

    best_feasible_sol <- list(sol = ord_pop[[sel_feas_vec[1]]],
                              res_eval = ord_res[c("AVaR_span", "mean_span",
                                                   "min_span", "max_span"),
                                                 sel_feas_vec[1]])
} else {
    best_feasible_sol <- list(sol = NULL,
                              res_eval = NULL)
}

# trace
res_trace[["best_res"]][, iter] <- ord_res[, 1]
res_trace[["solutions"]][[iter]] <- ord_pop[[1]]
if(!is.null(best_feasible_sol[["sol"]])){
    res_trace[["best_feas_res"]][, iter] <- best_feasible_sol[["res_eval"]]
}
res_trace[["feas_solutions"]][[iter]] <- best_feasible_sol[["sol"]]
```

```
res_trace[["diversity"]][iter] <- length(unique(popul))
res_trace[["num_feasible"]][iter] <- length(sel_feas_vec)
res_trace[["full_eval_df"]] <- rbind(res_trace[["full_eval_df"]],
                                     full_df)
res_trace[["time_stamp"]][iter + 1] <- Sys.time()

# GA loop
while(iter <= iter_max){

    # selection phase 1: elite
    ind_elite <- ord_vec[seq_len(n_elite)]

    # selection phase 2: tournament
    ind_tourn <- tournament_sel(pop = popul, obj_vec = obj_vec,
                                tourn_size = 2, n_out = n_new)
    pop_tourn <- popul[ind_tourn]

    # adapting some results for this population (used later)
    res_eval[, -ind_elite] <- res_eval[, ind_tourn]
    unknown_tourn <- unknown[ind_tourn]

    # cross over phase (only for population from tournament selection)

    # cross_over phase for prios
    sel_vec_cr_p <- sample(cross_sel_prio)
    pop_tourn[sel_vec_cr_p] <- cross_prio_pop(pop_tourn[sel_vec_cr_p])
    unknown_tourn[sel_vec_cr_p] <- TRUE

    # cross_over phase for modes
    sel_vec_cr_m <- sample(cross_sel_mode)
    pop_tourn[sel_vec_cr_m] <- cross_mode_pop(pop_tourn[sel_vec_cr_m])
    unknown_tourn[sel_vec_cr_m] <- TRUE

    # mutation prio
    sel_vec_mu_p <- sample(c(TRUE, FALSE), prob = c(p_muta_prio, 1-p_muta_prio),
                           size = n_new, replace = TRUE)
    if(any(sel_vec_mu_p)){
        for(i in which(sel_vec_mu_p)){
            pop_tourn[[i]] <- muta_prio(pop_tourn[[i]])
            unknown_tourn[i] <- TRUE
        }
    }

    # mutation mode
    sel_vec_mu_m <- sample(c(TRUE, FALSE), prob = c(p_muta_mode, 1-p_muta_mode),
                           size = n_new, replace = TRUE)
    if(any(sel_vec_mu_m)){
        for(i in which(sel_vec_mu_m)){
            pop_tourn[[i]] <- muta_mode(n_modes = project[["modes"]],
                                        sol = pop_tourn[[i]])
            unknown_tourn[i] <- TRUE
        }
    }

    # construction of new population and identification of cases with unknown obj_fun
    popul[-ind_elite] <- pop_tourn
    unknown[-ind_elite] <- unknown_tourn

    # adapting iteration number and nr_const
    iter <- iter + 1
    nr_const <- nr_const_fun(iter = iter, iter_max = iter_max)
    if(length(nr_const) == 1) nr_const <- rep(nr_const, times = 2)
    if(length(nr_const) > 2) nr_const <- nr_const[1:2]

    # evaluation of unknown cases and actualisation of obj_vec and ord_vec
    res_eval[, unknown] <- pop_eval_tr(pop = popul[unknown], paral = paral,
                                       project = project,
                                       alpha = alpha, dist = dist,
                                       nr_const = nr_const,
                                       lib.loc = lib.loc, cl = cl,
                                       det = det)

    # adapting the "known" cases to the current values of nr_const
    res_eval["obj_fun", !unknown] <- c(
        res_eval["AVaR_span", !unknown] +
            nr_const[1] * res_eval["n1_exceed", !unknown] +
            nr_const[2] * res_eval["n2_exceed", !unknown])
    res_eval["n1_const", !unknown] <- nr_const[1]
    res_eval["n2_const", !unknown] <- nr_const[2]
```

```
        obj_vec <- res_eval["obj_fun",]
        ord_vec <- order(obj_vec)
        ord_res <- res_eval[, ord_vec]
        ord_pop <- popul[ord_vec]

        full_df <- data.frame(ID = sapply(popul, FUN = sol_to_ID),
                              t(res_eval))

        # reset unknown cases (all known now)
        unknown <- rep(FALSE, times = n_pop)

        # adapting the best solution if necessary
        sel_feas_vec <- which(colSums(ord_res[c("n1_exceed", "n2_exceed"),]) == 0)

        if(length(sel_feas_vec) > 0){
            if(!is.null(best_feasible_sol[["sol"]])){
                if(best_feasible_sol[["res_eval"]]["AVaR_span"] >
obj_vec[ord_vec][sel_feas_vec[1]]){

                    best_feasible_sol[["sol"]] <- ord_pop[[sel_feas_vec[1]]]
                    best_feasible_sol[["res_eval"]] <- ord_res[c("AVaR_span",
                                                                 "mean_span",
                                                                 "min_span",
                                                                 "max_span"),
                                                               sel_feas_vec[1]]
                }
            } else {
                best_feasible_sol[["sol"]] <- ord_pop[[sel_feas_vec[1]]]
                best_feasible_sol[["res_eval"]] <- ord_res[c("AVaR_span",
                                                             "mean_span",
                                                             "min_span",
                                                             "max_span"),
                                                           sel_feas_vec[1]]
            }
        }

        # trace
        res_trace[["best_res"]][, iter] <- ord_res[, 1]
        res_trace[["solutions"]][[iter]] <- ord_pop[1]
        if(!is.null(best_feasible_sol[["sol"]])){
            res_trace[["best_feas_res"]][, iter] <- best_feasible_sol[["res_eval"]]
        }
        res_trace[["feas_solutions"]][[iter]] <- best_feasible_sol
        res_trace[["diversity"]][iter] <- length(unique(popul))
        res_trace[["num_feasible"]][iter] <- length(sel_feas_vec)
        res_trace[["full_eval_df"]] <- rbind(res_trace[["full_eval_df"]],
                                             full_df)
        res_trace[["time_stamp"]][iter + 1] <- Sys.time()
    }

    # close clusters if necessary
    if(cluster_out == TRUE) stopCluster(cl)

    # result
    list(best_feasible_sol = best_feasible_sol,
         trace = res_trace,
         parameters = list(project = project, iter_max = iter_max, n_pop = n_pop,
                           p_elite = p_elite, alpha = alpha, dist = dist,
                           p_cross_mode = p_cross_mode, p_cross_prio = p_cross_prio,
                           p_muta_mode = p_muta_mode, p_muta_prio = p_muta_prio,
                           nr_const_fun = nr_const_fun,
                           paral = paral))
}
```