Matlab-basierte Implementierung von heuristischen Lösungsverfahren sowie Entwurf und Implementierung einer Testumgebung für das Laubharkproblem

Ein Projektbericht von

Andreas H a r t m a n n

durchgeführt vom

April 2020

bis zum

Februar 2021

an der Fachhochschule Bielefeld

im Rahmen des Moduls "Biomechatronisches Projekt" des Masterstudiengangs BioMechatronik

Table of Contents

Einführung

Beschreibung des Laubharkproblems

Das Laubharkproblem ist ein kombinatorisches Optimierungsproblem, das sich an dem realen Vorgang des Laubharkes als ein spezielles Entsorgungsproblem orientiert. Ausgehend von einem Garten, der als beliebig fein gerasterte Matrix dargestellt wird, gilt es, das im Garten verteilte Laub auf möglichst kosteneffiziente Art und Weise auf einem Kompost zu entsorgen. Kosten entstehen dabei durch:

- Das Harken von Laub von einem Feld zum nächsten,
- Das Abtransportieren der Laubhaufen zu einem Kompost, sowie
- Das Zurücklegen unproduktiver Wege, die weder dem Harken noch dem direkten Abtransport dienlich sind.

Bei der Lösung des Laubharkproblems ergeben sich mehrere Fragestellungen (Wörtlich übernommen aus [1]):

- Soll das zu entsorgende Laub auf viele kleine oder wenige große Laubhaufen zusammengeharkt werden?
- Wo sollen diese Laubhaufen gebildet werden?
- Von welchen Feldern soll zu welchen Laubhaufen geharkt werden?
- In welcher Reihenfolge sollen dabei die einzelnen Felder bearbeitet werden?
- Welcher Aufwand ergibt sich dabei einerseits beim Harkprozess, andererseits beim Transportprozess (Abtransport der Laubhaufen mit einem Laubwagen zu einer oder zu mehreren Kompoststellen).
- Wie lassen sich dabei unproduktive Wege möglichst vermeiden?

Für eine detailliertere Einführung in die Thematik des Laubharkens, sowie eine Beleuchtung der mathematischen Grundlagen, wird der interessierte Leser auf das Manuskript von Hermann-Josef Kruse verwiesen [1], [2].

Beschreibung des Aufgabenfeldes

Im Rahmen des Projektes sind die folgenden Kerninhalte behandelt worden:

1. Implementierung heuristischer Lösungsverfahren nach Hermann-Josef Kruse
2. Entwurf und Implementierung einer Testumgebung für das Laubharkproblem

Darüber hinaus sind während der Arbeit an den Kerninhalten die folgenden Nebenaufgaben behandelt worden:

- Aufbereitung und Effizienzsteigerung des vorhandenen Quellcodes
- Schaffung und Vereinheitlichung von Interfaces zwischen Programmteilen
- Reduktion der Code-Komplexität durch Aggregation von verwandten Teilfunktionen in übergeordnete Funktionen
- Implementierung einer Klassenstruktur zur Objekt-orientierten Handhabung und Bedienung des Projekts
- Neugestaltung eines mit MATLABs veraltetem GUIDE erstellten GUI zur Nutzerinteraktion
- Vollständige Dokumentation der Quellcodes nach Pythons NumPy Standard

Dieses Liveskript stellt keine ausführliche Beschreibung aller im Detail vorgenommenen Veränderungen am Projekt dar. Der interessierte Nutzer sei zu diesem Zweck auf die Dokumentation der Quellcodes und die Git-Historie (von commit `d19c887` bis `d95c511`) des Projektes verwiesen. Sinn und Zweck dieses Skriptes ist es, den aktuellen Stand des Projektes zu dokumentieren und Nutzer, die mit der Arbeit am Laubharkproblem nicht vertraut sind, in die Bedienung der Quellcodes einzuführen. Die folgenden Unterkapitel widmen sich daher systematisch einzelnen Teilaspekten des Projektes, begonnen bei der Erzeugung eines Gartens zur Durchführung von Tests, über die Erstellung eines `ProblemData`-Objektes zur Verwaltung des Gartens samt einiger Metadaten und über die eingesetzten Algorithmen bis hin zum `TestManager`, mit dem parallelisiert Tests durchgeführt werden können.

Vorab sei angemerkt, dass es immer möglich und auch erwünscht ist, die Dokumentation der existierenden Codes zu Rate zu ziehen, um die genannten Beispiele nachzuvollziehen. Dazu wird der Befehl `help` oder `doc` gefolgt von dem Namen der Funktion in das *Command Window* eingegeben. Bei einfachen Skripten oder Matlab-Funktionen wird lediglich der Name angegeben. Bei Funktionen von Klassen (Methoden) muss der Klassenname gefolgt vom Methodennamen angegeben werden, wie folgt:

```
% Zeigt die Dokumentation im Command-Window an
help zufaelligerGarten
% Alternativ kann man sich die Dokumentation auch im Documentation-Browser ansehen
doc zufaelligerGarten

% Besonders interessant ist letzterer Fall fuer Klassen
doc TestManager
% Oder fuer eine explizite Methode
doc TestManager.runAllTests
```

Voraussetzungen

Wichtig: Bevor die Codes ausgeführt werden können, müssen sie in den aktuellen Matlab-Pfad aufgenommen werden. Dazu gibt es folgende Möglichkeiten:

- Man navigiert, nach dem Öffnen dieses Skriptes, in den Ordner `LHT/` des Projektes
- Man führt aus dem Ordner, in dem das Skript liegt, folgendem Befehl im Command-Window aus: `addpath(genpath("../../LHT/"));`
- Man fügt den LHT-Ordner von Hand aus der Seitenleiste in den Matlab Pfad ein (Muss rekursiv, also mit Unterordnern, eingefügt werden!)

Außerdem wird folgendes benötigt:

- MATLAB in Version 2019b **oder neuer**
- Parallel Computing Toolbox

Eine Reise durch den Quellcode

Der Quellcode ist zum Zeitpunkt der Abgabe dieser Projektarbeit (git commit `d95c511`) wie folgt organisiert:

```
Bienen/                                    Der Bienenalgorithmus aus der Masterarbeit von Julia Wieb
docs/                                      Dokumentation des Projektes
Ergebnisse/                                Ein Ordner mit einigen Testergebnissen für das "testScrip
Genetic/                                   Der genetische Algorithmus von Karim Abdelhak
RatingFunctions/                           Funktionen zur Berechnung des Gesamtaufwandes für eine Lö
Test/                                      Enthält einige Testfunktionen
Utility/                                   Sammlung aller Kernkomponenten des Projektes
Verfahren/                                 Sammlung aller implementierten Heuristischen Lösungsverfa
functionSignatures.json                    Eine unvollständige Datei zur Tab-Vervollständigung in MA
testFile.m                                 Matlab Skript zur Erprobung neuer Tests vor der Aufnahme
LHT.mlapp                                  Ein GUI zur grundlegenden grafischen Bedienung der Algori
testScript.mlx                             Ein LiveSkript mit einer Sammlung von Tests zum Vergleich
Variablen_und_Funktionsverzeichnis_LHT.mlx Unvollständige Dokumentation der im Projekt verwendeten P
```

Die Kommentare im folgenden Abschnitt dienen lediglich dazu, die Warnung von MATLABs Code-Analyzer zu unterdrücken

```
%#ok<*ASGLU>
%#ok<*NASGU>
```

Erzeugung eines Gartens

Ausgehend von der Theorie zum Laubharkproblem ist ein Garten ein ungerichteter, kanten und knotenbewerteter Graph. In Anlehnung an den Vorgang des realen Laubharkens ist in diesem Skript jedoch von *Feldern*, oder *Zellen* die rede. Mathematisch betrachtet sind immer Knoten gemeint. Ein Garten ist nun eine 2-dimensionale Matrix, die beschreibt, wie viel Laub auf je einem rechteckigen Abschnitt (*Feld, bzw. Knoten*) des Gartens liegt. Die Werte der Felder des Gartens haben die folgende Bedeutung:

- Feld > 0: Auf diesem Feld liegt die angegebene Menge Laub
- Feld $= 0$: Auf diesem Feld liegt kein Laub
- Feld $= (-1)$: Dieses Feld ist blockiert, durch einen Gartenschuppen
- Feld $= (-2)$: Dieses Feld ist blockiert, durch den Baumstamm eines Baums
- Feld $= (-10)$: Dieses Feld ist blockiert, hier steht der Kompost

Das Problem ist so ausgelegt, dass durch blockierte Felder ("Laubmenge" < 0) nicht gegangen und auch nicht geharkt werden kann. Die Kennzeichnung dient vor allem der Unterscheidbarkeit zwischen blockierten Feldern. So ist zum Beispiel immer klar, wo in einem Garten das Kompostfeld ist ("Laubmenge" $= (-10)$). Die Bewertungsfunktionen gehen davon aus, dass nur ein Kompost in einem Garten existiert, die Algorithmen hingegen interpretieren die negativen Laubmengen nicht weiter. Insofern können beliebige weitere Markierungen für blockierte Felder definiert werden.

Um einen Garten zu erzeugen, wird die Funktion `zufaelligerGarten()` aufgerufen. Als Parameter erwartet die Funktion wenigstens die Anzahl an Zeilen und Spalten des gewünschten Gartens. Der folgende Aufruf erzeugt also einen Garten mit 15 Zeilen und 15 Spalten:

```
Garten = zufaelligerGarten(15, 15)
```

Mit Hilfe weiterer Schlüssel-Wert (*Key-Value*) Parameter kann die Funktion modifiziert werden. Der Parameter `'Export'` speichert den erzeugten Garten in einer Datei ab, der Parameter `'Plot'` erzeugt zu dem Garten einen heatmap plot:

```
zufaelligerGarten(15, 15, "Plot", true)
```

In dem Plot sind die Werte der blockierten Felder mit 10 multipliziert, um sie gegenüber Feldern mit Laubmengen nahe 0 hervorzuheben. Der Kompost hat in dieser Ansicht den Wert NaN und ist schwarz eingefärbt. In der Gartenmatrix lauten die Werte wie oben beschrieben. Bei der Erzeugung des Gartens wird darauf geachtet, dass keine isolierten Laubfelder existieren können. Sollte ein isoliertes Feld vorhanden sein, wird der Garten so lange neu erstellt, bis alle laubbesetzten Felder erreichbar sind.

Um mit dem Garten arbeiten zu können, brauchen wir einige Metadaten:

- Einen Startknoten, an dem wir mit der Arbeit beginnen
- Eine maximale Laubmenge, die auf einem Feld aufgehäuft werden kann
- Die maximale Transportkapazität der Schubkarre
- Die Kostenfaktoren für das Harken, Transportieren und unproduktive Wege
- Eine Beschreibung des Gartens (`GMatrix`)
- Eine Beschreibung der Nachbarschaften der Felder (`Adjancency`)
- Die Distanzen der Felder zu einander, unter Berücksichtigung blockierter Felder (`DMatrix`)

Die `GMatrix` ist ein 4-spaltiger Vektor, dessen erste und zweite Spalte jeweils die Zeile und Spalte eines Gartenfeldes angeben. Die dritte Spalte definiert den Index des Feldes bei zeilenweiser Zählung, und die vierte Spalte beinhaltet die Laubmenge des Feldes. Die Adjazenzmatrix (`Adjacency`) beschreibt, welche Felder,

unter Berücksichtigung blockierter Felder, direkt mit einander benachbart sind. Die Distanzmatrix (`DMatrix`) beinhaltet die Entfernungen aller Felder zu einander, berechnet nach Dijkstras Algorithmus.

Um alle Metadaten zusammen mit einem zugehörigen Garten zu verwalten, gibt es die `ProblemData`-Klasse.

Die Beschreibung eines Problems: Garten mit Metadaten

Die `ProblemData`-Klasse entstammt ursprünglich Karim Abdelhaks genetischem Algorithmus und ist im Rahmen des Projektes adaptiert und erweitert worden, um eine allgemeine Verwendbarkeit mit allen Algorithmen zu ermöglichen. Ein Objekt vom Typ `ProblemData` stellt den Garten samt zugehöriger Metadaten zur Verfügung. Darüber hinaus sind weitere, algorithmen-spezifische Metadaten für die bioinspirierten Algorithmen hinterlegt. Auf diese Weise beschreibt ein Objekt der `ProblemData`-Klasse immer vollständig eine konkrete Problemstellung und ist die zentrale Datenstruktur, mit der alle Algorithmen arbeiten.

Um eine einfache Erzeugung von randomisierten Problemen zu ermöglichen, kann der zu verwendende Garten von der `ProblemData`-Klasse zufällig erzeugt werden. Dazu gibt man, wie zuvor, die gewünschte Anzahl an Zeilen und Spalten an:

Die ProblemData-Klasse

```
ProblemData(15, 15)
```

Um einen zuvor erstellten Garten in ein Objekt vom Typ `ProblemData` einzubetten, wird folgende Syntax verwendet:

```
pdata = ProblemData(20, 20, "Garten", Garten)
```

Die ersten beiden Parameter sind auch hier zwingend erforderlich und beschreiben, wie bei `zufaelligerGarten`, die Anzahl Zeilen und Spalten der Gartenmatrix. Bei dieser Art des Aufrufs werden die Zeilen und Spalten aber objektintern mit den entsprechenden Werten für den übergebenen Garten überschrieben.

Darüber hinaus bietet die `ProblemData`-Klasse viele weitere Parameter, die der Nutzer einstellen kann. Dabei gilt es zu beachten, dass eine Manipulation der enthaltenen Gartenmatrix (`pdata.Original`) in einem erzeugten Objekt **nicht erlaubt** ist. Eine derartige Manipulation würde nämlich automatisch zu einer Änderung der Adjazenz, Distanz- und GMatrix führen. In diesem Fall ist es besser, mit dem manipulierten Garten ein neues Objekt vom Typ `ProblemData` zu erzeugen.

Zwei Parameter in der Ausgabe sollen noch besonders hervorgehoben werden. Der Parameter `DiagonalWeight` bestimmt die Kosten (als Wegeinheit) für das diagonale Gehen. Ein Wert von `inf` verbietet das diagonale Gehen. Ein Wert von $\sqrt{2}$ entspräche der erwarteten euklidischen Distanz. Der Parameter `flattened` ist nicht vom Nutzer parametrierbar, sondern von der Struktur fest vorgegeben. Da die Metadaten Adjazenz-, Distanz- und GMatrix aus der Gartenmatrix berechnet werden können, müssen diese nicht zwingend abgespeichert werden. Grade für große Gärten von ca. 70 x 70 Feldern ist der Speicherbedarf dieser Strukturen um einen Faktor 500 größer als der Speicherbedarf der anderen in `ProblemData` enthaltenen Parameter.

Daher können Objekte vom Typ `ProblemData` "komprimiert" werden, wobei die Adjazenz-, Distanz- und GMatrix gelöscht werden. Diese Komprimierung findet mit der Methode `flatten` statt:

Reduzierung des Speicherbedarfs von ProblemData

```
pdata = pdata.flatten()
```

In diesem Zustand können die Algorithmen das Problem nicht lösen, da essenzielle Metadaten fehlen. Daher ist es nötig, ein komprimiertes Objekt vom Typ `ProblemData` vorher zu "dekomprimieren":

```
pdata = pdata.unflatten()
```

Dabei werden die fehlenden Metadaten von neuem berechnet und eingesetzt.

Weitere Methoden von ProblemData

Des weiteren kann die enthaltene Gartenmatrix direkt geplottet werden:

```
pdata.plot()
```

oder man lässt sich eine textuelle Beschreibung des Problems ausgeben, zum Beispiel im Command Window:

```
pdata.print()
```

oder in einer Textdatei :

```
fd = fopen("beispielaufruf_pdata_print.txt", "w");
pdata.print(fd);
fclose(fd);
```

HIerbei ist jedoch zu beachten, dass Algorithmen-spezifische Parameter nicht mit ausgegeben werden.

Die `ProblemData`-Klasse akzeptiert noch viele Konstruktor-Key-Value-Parameter, um sehr spezifische Probleme erzeugen zu können. Mit dieser Problembeschreibung als Grundlage können wir fortfahren und die Algorithmen zur Lösung der Probleme betrachten.

Heuristische Lösungsverfahren

Die heuristischen Lösungsverfahren, enthalten im Ordner `Verfahren/`, erzeugen auf deterministische Art und Weise Lösungen für das Laubharkproblem. Dabei lassen sich zwei übergeordnete Algorithmentypen unterscheiden: Die simultanen Clusterverfahren und die sukzessiven Clusterverfahren. Die weiteren im Ordner `Verfahren/` enthaltenen Funktionen stammen aus dem Projekt vorangegangenen Überlegungen zur Lösung des Laubharkproblems. Jedes der Verfahren ist samt seiner Funktionsweise im Quellcode dokumentiert,

daher wird an dieser Stelle auf eine Erörterung der Algorithmen verzichtet. Der interessierte Leser wird auf die Beschreibungen von Hermann-Josef Kruse verwiesen.

Alle Verfahren haben ein fest definiertes Interface: Sie erhalten ein Objekt vom Typ `ProblemData` und erzeugen daraus eine Nachfolgerfunktion `s`, die die ermittelte Harkvorschrift zur Lösung des Problems beschreibt. Da die Grundalgorithmen (`ho, co, simultaneous_cluster, successive_cluster`) noch weitere Parameter zur Festlegung interner Entscheidungskriterien erhalten, sind für jeden der Grundalgorithmen spezifische Wrapper-Funktionen implementiert worden, die die Anforderungen an das Interface erfüllen.

Zum Zeitpunkt der Projektabgabe umfasst das Laubharkprojekt 87 verschiedene Heuristiken. Zur Ermittlung, welche der Heuristiken besonders gute Ergebnisse liefern, ist ein eigenes Testszenario entwickelt worden, dass im `testScript` zu finden ist.

Eine Heuristik lässt sich zwar direkt aufrufen:

```
s = co_max(pdata)
```

Allerdings ist diese Art des Aufrufs nicht vorgesehen. Um eine Kompatibilität zu den genetischen Algorithmen herzustellen, die unter anderem weitere Rückgabeparameter als die Heuristiken liefern, gibt es die Wrapper-Klasse `LHT_HeuristicData`, die von der Klasse `LHT_BaseData` erbt. Daher sollten die Heuristiken in die Wrapper-Klasse "gewickelt" werden, die zusätzliche Komfortfunktionen liefert, wie gleich ersichtlich wird:

Die LHT_HeuristicData-Klasse

```
lht_co_max = LHT_HeuristicData(@co_max, "Cluster_Max")
```

Um eine Berechnung durchzuführen, wird ein `ProblemData` Objekt übergeben:

```
[index, result] = lht_co_max.add_new_data(pdata)
```

Die Variable `result` enthält ein Struct mit den Ergebnissen der Berechnung. Alle auf diese Weise durchgeführten Berechnungen werden Objekt-intern gespeichert. Der Index `index` gibt einerseits an, wie viele Ergebnisse bereits gespeichert worden sind. Auf der anderen Seite kann mit diesem Index das gewünschte Ergebnis aus der internen Ergebnisetabelle ermittelt werden:

Datenzugriff in LHT_HeuristicData

```
table_row = lht_co_max.get(index)
```

Ein Aufruf der `get()` Methode ohne Parameter liefert die gesamte Tabelle zurück. Dazu füllen wir erst noch einige Ergebnisse ein:

```
for idx = 1:5
    lht_co_max.add_new_data(ProblemData(15+idx, 15+idx));
end
lht_co_max.get()
```

Alternativ lassen sich auch gezielte Bereiche ausgeben:

```
lht_co_max.get([2, 4])
```

Um sich die Anzahl an Ergebnissen in der Tabelle nicht merken zu müssen, kann mit `Inf` als Ersatz für das "end" in Arrays das letzte Element der Tabelle indexiert werden:

```
lht_co_max.get(Inf)
```

Da oftmals nur ein Bruchteil der Daten einer so erhaltenen Tabellenzeile benötigt wird, kann man die zu extrahierenden Daten weiter spezifizieren:

```
lht_co_max.get(2, "Kosten")
```

Welche Daten genau auf diesem Wege extrahiert werden können, verrät die Dokumentation zu der Methode:

```
help lht_co_max.get
```

Laubmengenoptimierung der Heuristiken

Um den Heuristiken einen potenziellen Vorteil für Problemstellungen zu verschaffen, in denen das Verhältnis der maximalen Laub-Transportmenge und der maximalen Laubmenge pro Feld ungünstig gewählt sind, kann die Laubmenge der Heuristiken optimiert werden. Bei dieser Laubmengenoptimierung durchlaufen die Heuristiken beim Lösen des Gartens eine Schleife, in der das Problem mit jeder Laubmenge im Intervall $[1, MaxLaub]$ gelöst und die minimale Lösung ermittelt wird. Das so ermittelte Laubmengenoptimum wird als zusätzlicher Wert in den "results"-Strukturen eingepflegt, damit die Ergebnisse nachgerechnet werden können.

Um die Laubmengenoptimierung zu aktivieren, muss ein zusätzlicher Konstruktorparameter übergeben werden:

```
LHT_HeuristicData(@co_max, "Cluster_Max", "OptimizeMaxLaub", true);
```

Weitere Methoden von LHT_HeuristicData

Um die errechneten Daten schnell über der Gartengröße zu plotten, bietet sich die `plot`-Methode an:

```
figure();
ax = axes();
lht_co_max.plot(ax, "Kosten")
```

Analog zur `print`-Methode von `ProblemData`, kann ein Objekt vom Typ `LHT_HeuristicData` seine Lösung ebenfalls textuell darstellen. Dazu muss zwingend der gewünschte Durchlauf (index) angegeben werden:

```
lht_co_max.print(2)
```

Um nach einem Durchlauf alle vorhandenen Ergebnisse zu löschen und von neuem Berechnungen zu beginnen, wird die `clear` Methode verwendet:

```
lht_co_max.clear()
lht_co_max.add_new_data(pdata)
```

Zuletzt kann zum Beispiel zur Erzeugung von Legendeneinträgen in Plots der eingangs übergebene Name des Algorithmus formatiert ausgegeben werden:

```
lht_co_max.get_name("Latex")
```

Darüber hinaus gibt es weitere Konstruktor- und Methodenparameter. Auch hier wird der interessierte Leser auf die Dokumentation der Methode verwiesen.

Komfortable Erzeugung von LHT_HeuristicData-Objekten

Da, wie eingangs erwähnt, 87 Heuristiken im Laubharkprojekt existieren, die alle ihrerseits unterschiedliche Funktionspointer verwenden und unterschiedliche Bezeichner haben, ist eine Komfortmethode integriert worden, die die schnelle und einfache Erzeugung von `LHT_HeuristicData`-Objekten erlaubt. Die Methode `gather()` arbeitet auf Basis der Textdatei `Verfahren/heuristics.txt`, die für jede Heuristik eine textuelle Beschreibung des Funktionshandle und Algorithmennamen beinhaltet. Die Methode `gather()` liest, basierend auf den Eingaben des Nutzers, die gewünschten Algorithmen aus der Datei aus und erzeugt dafür Objekte vom Typ `LHT_HeuristicData`. Dazu muss der Nutzer lediglich die ID des gewünschten Algorithmus kennen, die der Textdatei zu entnehmen ist:

```
all_heuristics = LHT_HeuristicData.gather("Range", (1:end))
```

Um im "großen Stil" mit den Algorithmen zu arbeiten, können der `gather()` Methode auch Konstruktorparameter übergeben werden, die auf alle erzeugten Objekte angewendet werden:

```
LHT_HeuristicData.gather("OptimizeMaxLaub", true, "StoreResults", false, "NameSuffix",
```

Bio-inspirierte Lösungsverfahren

Zum Zeitpunkt des Projektberichts existieren zwei bio-inspirierte Lösungsverfahren im Laubharkprojekt: Der genetische Algorithmus von Karim Abdelhak (im Ordner `Genetic/`) und der Bienenschwarm-Algorithmus von Julia Wiebe (im Ordern `Bienen`). Auf eine Erklärung der Funktionsweise der Algorithmen soll auch hier verzichtet werden. Stattdessen wird der Nutzer auf die externe Dokumentation hingewiesen.

Wie bereits erwähnt haben die bio-inspirierten Lösungsverfahren ein anderes Interface als die Heuristiken. Dies ist insbesondere der Tatsache geschuldet, dass die hier implementierten bio-inspirierten Verfahren mit Populationen von Individuen arbeiten und in diesem Sinne mehr Rückgabeparameter als nur eine Nachfolgerfunktion *s* liefern können. Um diesem Umstand gerecht zu werden, stellt die Wrapper-Klasse `LHT_GeneticData` dem Nutzer eine Datenkapselung bereit, mit der die bio-inspirierten Algorithmen auf die gleiche Weise wie die Heuristiken in `LHT_HeuristicData` verwendet werden können. Hierdurch werden Implementierungsunterschiede vor dem Nutzer verborgen und er kann mit einem einheitlichen Interface arbeiten.

Daher sind viele der im Folgenden demonstrierten Methoden identisch zu denen der Klasse `LHT_HeuristicData`. Ein Objekt eines bio-inspirierten Verfahren erzeugt man wie folgt:

Die LHT_GeneticData-Klasse

```
% Fuer den Bienenalgorithmus:
% lht_bio = LHT_GeneticData(@bienenalgorithmus, "Bienen-Algorithmus")
lht_bio = LHT_GeneticData(@genetic, "Genetischer-Algorithmus");
```

Anders als bei den Heuristiken, können dem Konstruktor diverse Parameter übergeben werden:

```
help LHT_GeneticData.LHT_GeneticData
```

Um eine Berechnung durchzuführen, wird auch hier ein `ProblemData`-Objekt übergeben:

```
pdata.Popsize = 15;  % Damit der genetische Algorithmus nicht so lange rechnet...
[index, result] = lht_bio.add_new_data(pdata)
```

Die Variable `result` enthält ein Struct mit den Ergebnissen der Berechnung. Alle auf diese Weise durchgeführten Berechnungen werden objekt-intern gespeichert. Der Index `index` gibt einerseits an, wie viele Ergebnisse bereits gespeichert worden sind. Auf der anderen Seite kann mit diesem Index das gewünschte Ergebnis aus der internen Ergebnisetabelle ermittelt werden:

Datenzugriff in LHT_GeneticData

```
table_row = lht_bio.get(index)
```

Ein Aufruf der `get()`-Methode ohne Parameter liefert die gesamte Tabelle zurück. Dazu füllen wir erst noch einige Ergebnisse ein:

```
% Dieser Schritt dauert eine ganze Weile.
% Mit dem Konstruktorparameter "NumWorkers" liese er sich durch parallelisierung
% erheblich Beschleunigen.
for idx = 1:5
    lht_bio.add_new_data(ProblemData(15+idx, 15+idx, "PopulationSize", 15));
end
lht_bio.get()
```

Alternativ lassen sich auch gezielte Bereiche ausgeben:

```
lht_bio.get([2, 4])
```

Um sich die Anzahl an Ergebnissen in der Tabelle nicht merken zu müssen, kann mit `Inf` als Ersatz für das "end" in Arrays das letzte Element der Tabelle indexiert werden:

```
lht_bio.get(Inf)
```

Da oftmals nur ein Bruchteil der Daten einer so erhaltenen Tabellenzeile benötigt wird, kann man die zu extrahierenden Daten weiter spezifizieren:

```
lht_bio.get(2, "Kosten")
```

Anders als bei den Heuristiken ist hierbei zu beachten, dass für jeden Durchlauf der Monte-Carlo-Simulation ein einzelnes Ergebnis vorliegt. Wird die Methode `get()` ohne dritten Parameter aufgerufen, wird das "Beste" Ergebnis (geringste Gesamtkosten) aus allen Durchläufen ausgegeben. Um die Ergebnisse aller Durchläufe zu sehen:

```
lht_bio.get(2, "Kosten", "All");
```

Welche Daten genau auf diesem Wege extrahiert werden können, verrät auch hier die Dokumentation zu der Methode:

```
help lht_bio.get
```

Weitere Methoden von LHT_GeneticData

Um die errechneten Daten schnell über der Gartengröße zu plotten, bietet sich die `plot`-Methode an:

```
fig = figure();
ax = axes();
lht_bio.plot(ax, "Kosten")
```

Man beachte, dass analog zur `get()`-Methode auch hier ein dritter Parameter angegeben werden kann, um z.B. die Mittelwerte aus allen Monte-Carlo-durchläufen zu plotten:

```
lht_bio.plot(ax, "Kosten", "Average")
```

Analog zur `print`-Methode von `ProblemData`, kann ein Objekt vom Typ `LHT_GeneticData` seine Lösung ebenfalls textuell darstellen. Dazu muss zwingend der gewünschte Durchlauf (index) angegeben werden:

```
lht_bio.print(2)
```

Die `print`-Methode verwendet in der Ausgabe für jeden Kostenfaktor den Wert der Lösung mit den geringsten Gesamtkosten. Eine Modifikation dieses Verhaltens muss daher direkt im Quelltext der Methode erfolgen.

Um nach einem Durchlauf alle vorhandenen Ergebnisse zu löschen und von neuem Berechnungen zu beginnen, wird die `clear`-Methode verwendet:

```
lht_bio.clear()
lht_bio.add_new_data(pdata)
```

Zuletzt kann zum Beispiel zur Erzeugung von Legendeneinträgen in Plots der eingangs übergeben Name des Algorithmus formatiert ausgegeben werden:

```
lht_bio.get_name("Latex")
```

Durchführung von Tests zum Vergleich von Algorithmen: Der TestManager

Der `TestManager` ist die Hauptkomponente der Projektarbeit. Seine Aufgabe ist es, jeden durchzuführenden Test mit jedem zu testenden Algorithmus auszuführen und die Ergebnisse dem Nutzer zugänglich zu machen. Dazu hat der `TestManager` eine interne Tabelle, in der die Testszenarien die Zeilen und die zu testenden Algorithmen die Spalten darstellen. Er erfüllt zum aktuellen Zeitpunkt die folgenden Aufgaben:

- Aufnahme beliebig vieler zu testender Algorithmen
- Aufnahme beliebig vieler durchzuführender Testszenarien
- Parallelisierte Durchführung aller vorbereiteten Tests
- Zugriff auf den gesamten Ergebnisdatensatz
- Zugriff auf bestimmte Teile des Ergebnisdatensatzes (Filterung der Daten)
- Speicherung und spätere Wiederherstellung des TestManagers mit Matlabs `save` und `load`

Durch das Code-Design ist es beispielsweise möglich, eine zuvor gespeicherte Instanz des `TestManagers` wiederherzustellen und neue Tests und Algorithmen einzufügen und ausführen zu lassen. Dabei ist zu beachten, dass nur diejenigen Tests ausgeführt werden, die noch nicht vorher ausgeführt worden sind. Außerdem ist es möglich, den `TestManager` im Betrieb zu unterbrechen und zu einem späteren Zeitpunkt an der abgebrochenen Stelle mit dem Testen fortfahren zu lassen. All diese Anwendungsszenarien werden in den folgenden Beispielen gezeigt.

Der Konstruktor des `TestManagers` ist immer leer:

```
tm = TestManager()
```

Die Tests sind Objekte vom Typ `ProblemData`. Dabei stellt jedes Objekt einen eigenen durchzuführenden Test (Testszenario) dar. Nun fügen wir einige zufällig erzeugte Tests ein:

```
for idx = 1:5
    pdata = ProblemData(15, 15, "PopulationSize", 15);
    info_str = sprintf("Testcase %d", idx);
    tm.addTestCase(pdata, info_str);
end
```

Das zweite Argument der Funktion ist ein optionaler String, mit dem das jeweilige Testszenario beschrieben werden kann. Das macht es in der Ergebnistabelle leichter, den Überblick über die Bedeutung der Tests zu behalten. Die Methode `get_results()` zeigt uns die gesamte Ergebnistabelle an:

```
tm.get_results()
```

Hier sehen wir nun die eingefügten Testszenarien. Es fehlen noch einige zu testende Algorithmen:

```
for alg = LHT_HeuristicData.gather("Range", 1:5)
    tm.addAlgorithm(alg);
end
tm.addAlgorithm(LHT_GeneticData(@genetic, "Genetic"))
```

Rufen wir das Objekt in der Kommandozeile ohne Methode auf, erfahren wir, wie der Status aussieht:

```
tm
```

Aber wir können uns ebenso die Ergebnistabelle ansehen:

```
tm.get_results()
```

Sind die Vorbereitungen getroffen, kann das Testen beginnen. Wie bereits erwähnt werden die Tests parallelisiert ausgeführt. Spätestens hier ist also die Parallel Computing Toolbox erforderlich:

```
tm.runAllTests()
```

Wie man sieht, informiert uns der Testmanager während der Ausführung der Tests über den aktuellen Status. Der Ladebalken wird bewusst nur alle 2 Sekunden aktualisiert, um möglichst viele CPU-Ressourcen für das Ausführen der Tests übrig zu lassen.

Nun, da die Ergebnisse vorliegen, können wir anfangen, die Daten auszuwerten. Das geht entweder per Hand direkt in der Ergebnistabelle, oder wir nutzen die angebotenen Methoden zum Filtern der Daten. Dafür ist die Methode `extract()` zuständig. Die Methode ist komplex in der Bedienung, aber sehr mächtig hinsichtlich des Nutzen zur Datenauswertung. Am besten ist es, sich hier mit der Dokumentation vertraut zu machen, bevor das nächste Beispiel eingeführt wird:

```
help tm.extract
```

Nehmen wir an, wir möchten die Gesamtkosten des Algorithmus "Zickzack" für jeden Garten in jedem Test ermitteln.

- Der Algorithmus ist "Zickzack".
- Die Variable ist "Kosten", hier "K".
- Wir sortieren die Daten nicht, weil es für ein und den selben Garten jeweils nur einen Test und damit nichts zum sortieren gibt.

- Wir gruppieren die Daten nach den verschiedenen Gärten (Bezeichner für Gärten in ProblemData ist "Original").

Der Aufruf sieht dann wie folgt aus:

```
[data, grouped_by] = tm.extract("Zickzack", "K", "GroupBy", "Original")
```

Nun könnten wir die erhaltenen Daten direkt plotten. Die Daten in `data(1, 1, 3)` gehören zu dem Test des dritten Gartens, also `grouped_by{3}`. Die Selektion der Daten lässt sich, zum Beispiel mit dem Parameter "From", weiter verfeinern. So könnten wir die Ergebnisse spezifisch für Gärten abfragen, in denen der Index des Kompost (`ProblemData.Target`) kleiner als 100 ist:

```
selection = tm.where("Target", "<", 100)
[data, grouped_by] = tm.extract("Zickzack", "K", "GroupBy", "Original", "From", selecti
```

Weiterführende Beispiele und sonstige Anwendungen des TestManager sind in der Datei `testScript` zu sehen, in welcher der `TestManager` einige Testszenarien berechnet. Um den `TestManager` abzuspeichern, verwendet man Matlabs `save()`-Befehl.

```
save mytm_save tm
clear tm
load mytm_save
tm.get_results()
```

Wie man sieht sind nach dem Speichern, Löschen und erneuten Laden des Testmanagers alle Informationen noch erhalten.

Von den Erläuterungen zu `ProblemData` sollte bekannt sein, dass die Metadaten (Distanz-, Adjazenz- und GMatrix) den großen Teil des Speichers benötigen. Zu diesem Zweck lassen sich alle Metadaten der `ProblemData`-Objekte in der Tabelle vor dem Speichern entfernen:

```
tm.flatten();
save mytm_save_compressed tm
clear tm
load mytm_save_compressed
tm.unflatten();
tm.get_results()
```

Und nach dem Speichern wiederherstellen. Bei der Betrachtung der von Matlab erzeugten Speicherdateien sollte auffallen, dass die "komprimierte" Variante weniger Speicher braucht. Gerade bei großen Gärten von 50 x 50 Feldern und mehr fällt dieser Unterschied sehr groß aus. Abschließend ist zu zeigen, dass ein späteres Hinzufügen neuer Tests und Algorithmen kein Problem für den `TestManager` darstellt:

```
tm.addAlgorithm(LHT_HeuristicData.gather("Range", 12));
tm.addTestCase(ProblemData(20, 20, "PopulationSize", 15));
tm.runAllTests();
tm.get_results()
```

Anhang

Zusammenfassung aller am Projekt durchgeführten Erweiterungen und Veränderungen

In diesem Abschnitt wird in stichpunktartiger Form zusammengefasst, welche Änderungen jeweils am Projekt gegenüber dem ursprünglichen Zustand vorgenommen wurden. Viele Änderungen sind im AMMO Gitlab dokumentiert und mit jeweiligen Issues/Merge-Requests versehen. So lassen sich über Gitlab alle durchgeführten Änderungen nachvollziehen und einem konkreten Issue zuordnen. Einige Änderungen werden von den Issues noch nicht erfasst, weil sie zu weit zurück liegen. Diese Änderungen können über die Commit-Historie nachvollzogen werden.

Veränderungen am ursprünglichen Projekt

Alte/obsolete Projektdateien entfernt

Verzeichnisstruktur erzeugt

- Algorithmen wurde aus dem Hauptverzeichnis in diverse neue Unterverzeichnisse sortiert
- Unterverzeichnisse sind nach "Zweck" getrennt

Vereinheitlich von Inzidenzmatrizen zu Adjazenzmatrizen

Endlosschleife in genetischen Verfahren behoben

- Verursacht durch unzulässige Mutationen, die Schleifen erzeugten

Fehlerbeseitigung in Aufwandsberechnung des genetischen Algorithmus

- Berechnung einzelner Teilkosten war fehlerhaft

Korrekte Berechnung der Entfernungsmatrizen nach Dijkstras Algorithmus

- Zuvor: Manhattan-Metrik,
- Berücksichtigt nun Hindernisse zwischen Feldern
- Basiert auf Matlab `graph` Objekten, hocheffiziente Implementierung.

Verbesserte Funktion zur Erzeugung von Testgärten

- Erzeugt Gärten nach dem selben Schema wie zuvor
- Code wurde Vektorisiert und prüft auch auf Randfälle
- Erzeugte Gärten werden auf Gültigkeit geprüft und so lange neu erzeugt, bis sie gültig sind

- Gärten können optional in Dateien exportiert oder geplottet werden
- Code ist jetzt vollständig dokumentiert

Überarbeitung der existierenden Heuristiken

- Heuristiken wurden mit vektorisiertem Code neu implementiert (Effizienter)
- Codepassagen werden zwischen verwandten Algorithmen geteilt
- Fehler in Heuristiken wurden beseitigt und im direkten Vergleich mit Lösungen von Hermann-Josef Kruse auf Korrektheit geprüft
- Funktionsnamen wurden vereinheitlicht
- Code ist jetzt vollständig dokumentiert.

Vereinheitlichung der Datenstrukturen

- Vereinheitlichung der Übergabe/Rückgabeparameter für alle Heuristiken
- Entfernung globalen Variablen, die zum Datenaustausch genutzt wurden
- Einführung der ProblemData Klasse aus dem genetischen Algorithmus in allen Funktionen
- Entfernung unbenutzter "Zwischenstrukturen" (Qhilf, deltas, ...)

Aggregation von verwandten Teilfunktionen in eine Funktion

- Einführung der Funktion `analyze_s`, die die Arbeit von vorher 5 einzelnen Funktionen übernimmt
- Elimination temporärer Variablen, die lediglich als Zwischenergebnisse dienten um andere Strukturen zu berechnen

Vereinheitlichung der blockierten Felder

- Blockierte Felder haben einen Wert < 0
- Algorithmen prüfen blockierte Felder nun mit < 0, statt $= (-1)$ oder ähnlichem
- -1 Steht nun für den Gartenschuppen
- -2 Steht für Bäume im Garten
- -10 Steht für den Kompost

Vereinheitlichung der Start-/Kompostknoten

- Explizite Prüfung der Erreichbarkeit des Komposts
- Berechnung der unproduktiven Wege/Transportarbeit mit Pendeltouren

Flächiger Einsatz von ProblemData als Datenstruktur

- Einheitlicher Zugriff auf Garten-bezogene Daten
- Erweiterung um plot() und print() Methode, um Gärten zu inspizieren
- Einführung eines Konstruktors mit optionalen Key-Value Parametern
- Berechnung aller Strukturparameter (GMatrix, DMatrix, Adjacency) in ProblemData

Erweiterungen zum ursprünglichen Projekt

Neues GUI für das LHT mit Hilfe des Matlab AppDesigner

- Umfasst noch nicht alle Features, die das Projekt zum aktuellen Zeitpunkt bietet
- Zur Grundlegenden Einführung in das Projekt geeignet

Implementierung einer LHT_BaseData Klasse zur Verwaltung aller Algorithmen

- Einheitliches Interface für alle Algorithmen (Heuristiken, wie auch bio-inspirierte)
- Speichert die Ergebnisse von Berechnungen der Algorithmen
- Gespeicherte Daten können mit Methoden gefiltert ausgegeben werden
- Wird von Kindklassen für bestimmte Algorithmen spezialisiert:
- **LHT_HeuristicData**: Spezialisierung zum Verwalten der heuristischen Algorithmen
- **LHT_GeneticData**: Spezialisierung zum Verwalten der bio-inspirierten Algorithmen (Brauchen z.B. Monte-Carlo Simulation)

Implementierung einer TestManager Klasse zur Verwaltung und parallelen Ausführung von Tests

- Geeignet zur Durchführung aller Tests
- Verwaltet ProblemData und LHT_BaseData
- Führt Tests parallel aus, gibt textuelle Fortschrittsanzeige
- Speichert alle Testergebnisse in interner Tabelle
- Daten können mit speziellen Methoden gezielt extrahiert werden

Einführung des TestScript zur Durchführung aller Tests

- Alle durchzuführenden Tests werden hier zentral gesammelt

Implementierung diverser heuristischer Lösungsverfahren nach Hermann-Josef Kruse (siehe Anhang)

- Simultane Clusterverfahren
- Sukzessive Clusterverfahren

Bestimmung einer optimalen Laubobergrenze für heuristische Algorithmen

- Heuristische Lösungsverfahren lösen einen gegebenen Garten mit jeder Laubmenge im Interval [1, ProblemData.Max-Val]
- Die kostengünstigste Lösung wird jeweils übernommen

Integration des Bienenalgorithmus von Julia Wiebe

- Bio-inspirierter Algorithmus zur Lösung des Laubharkproblem
- Wurde initial mit "altem" Projektstand entwickelt, ist in das aktuelle Projekt eingepflegt worden

Implementierung der Hubzentrierungs-Heuristik nach Hermann-Josef Kruse

- Heuristik zur Nachbesserung von erzeugten Lösungen
- Siehe testScript.mlx, Test 1.4

Dokumentation der Quellcodes

- Dokumentation einheitlich nach NumPy Standard
- Automatisierte Erzeugung von HTML/PDF/EPUB Dokumentation mit Pythons Sphinx
- Erzeugung der Dokumentation ist auch containerisiert

Implementierungsdetails

Der Code in dieser Sektion sollte einmal ausgeführt werden, da die Hilfetexte zu den Funktionsparametern direkt aus den entsprechenden Matlab Funktionen extrahiert werden.

```
% Hilfsfunktion
helpfor = @(name, section) (...
    disp( ...
        subsref( ...
            regexp( ...
                help(name), ...
                sprintf("\\s*?%s.*?\\n\\s*?\\n", section), ...
                'match'), ...
            struct('type', '{}', 'subs', {{1}}))...
        ) ...
    );
```

Sukzessive Clusterverfahren

Bei dieser Klasse von Heuristiken wird zu jedem Zeitpunkt nur ein Cluster betrachtet, dass so gut wie möglich aufgefüllt wird, bevor das nächste Cluster eröffnet und betrachtet wird. In anderen Worten wird jedes eröffnete Cluster sofort aufgefüllt, die Algorithmen sind daher *greedy*.

Das Verhalten des Algorithmus wird über seine Übergabeparameter gesteuert:

```
helpfor("successive_cluster", "Key-Value Parameters")
```

Der grundlegende Ablauf ist wie folgt:

1. Es wird der Knoten mit dem kleinsten Index zum Repräsentanten für das Cluster ernannt
2. Alle Knoten in der Nachbarschaft werden auf "Fit-Kriterien" überprüft (Siehe [3])
3. Alle passenden Nachbarknoten werden in eine Liste (`CandidateListType`) aufgenommen
4. Knoten aus der Liste werden auf ihre Tauglichkeit zum hinzufügen in das aktuelle Cluster geprüft und ggf. aus der Liste entfernt

5. Die Liste wird so lange abgearbeitet und neu befüllt, bis das Cluster voll ist
6. Im so erzeugten Cluster wird ein Hub bestimmt (`ClusterSelection`)

Eine detailliertere Beschreibung ist der Funktionsdokumentation von '`successive_cluster.m`' oder den ursprünglichen Überlegungen von Hermann-Josef Kruse [3] zu entnehmen.

Simultane Clusterverfahren

Bei dieser Klasse von Heuristiken werden mehrere Cluster gleichzeitig betrachtet. Die Cluster werden nicht unmittelbar nach Ihrer Eröffnung aufgefüllt, sondern können auch über längere Zeiträume unberücksichtigt bleiben, oder gar nicht vollständig befüllt werden.

Das Verhalten dieser Heuristiken wird ebenfalls über Übergabeparameter gesteuert:

```
helpfor("simultaneous_cluster", "Key-Value Parameters")
```

Hier wird stets ein sogenannter "Zuweisungskandidat", sowie ein "Kontaktkandidat" betrachtet. Der Zuweisungskandidat ist dabei stets der aktuell betrachtete Knoten. Der Kontaktkandidat wird in der Nachbarschaft dieses Zuweisungskandidaten ausgewählt. Auch diese Heuristik zeichnet sich dadurch aus, dass die Auswahl eines Hubs im Cluster zunächst Implizit ist. Das Auswahlkriterium wird durch den Parameter '`HubStrategy`' festgelegt.

Der Ablauf lautet wie folgt:

1. Es wird ein Zuweisungskandidat aus der Liste der noch nicht betrachteten Knoten ausgewählt (`AssignmentCandidateSelection`)
2. In der Nachbarschaft dieses Knoten wird ein Kontaktkandidat ausgewählt (`ContactCandidateSelection`)
3. Auf Grundlage der (vorläufigen) Clusterzugehörigkeiten wird entschieden, ob ein neues Cluster erzeugt wird, oder ein bisheriges Cluster verworfen und in ein vorhandenes Cluster integriert wird.
4. Zuletzt wird bestimmt, welcher Knoten den Hub des Clusters darstellt.

Auch hier wird der Leser für eine detailliertere Beschreibung des Algorithmus auf die Dokumentation in der Funktion selbst, oder die Überlegungen von Hermann-Josef Kruse [3] verwiesen.

Der Bienenalgorithmus

Der Bienenalgorithmus wurde im Rahmen einer Masterarbeit von Julia Wiebe implementiert. Er basiert auf dem natürlichen Prinzip des Schwänzeltanzes von Bienen. Da der Algorithmus ursprünglich mit einer über einem

Jahr alten Version des Laubharkprojektes entwickelt worden ist, wurde er in den aktuellen Stand des Projektes integriert.

Bei der initialen Portierung ist nicht berücksichtigt worden, dass die *Scout Bienen* den Lösungsraum randomisiert durchsuchen. Stattdessen erzeugten die Scout Bienen in jeder Iteration neue Lösungen mit Hilfe einer festen Menge an deterministischen Heuristiken. Dadurch trugen die Scout-Bienen nicht zur Verbesserung der Lösungen bei, da bereits nach einer geringen Anzahl an Iterationen alle Heuristiken mindestens einmal aufgerufen, und damit die besten derart auffindbaren Lösungen bereits gefunden waren. Als Reaktion darauf ist ein Algorithmus `random_s` entwickelt worden, der randomisierte Lösungen erzeugt. Zum Zeitpunkt der Abgabe ist eine Integration dieser Funktion, ebenso wie ein vergleichender Test, noch ausstehend.

Von allen implementierten Algorithmen ist der Bienenalgorithmus der langsamste. Allerdings findet der Bienenalgorithmus absolut gesehen die besten Lösungen. Die Performance des Algorithmus ist in erster Linie dadurch zu erklären, dass jeder zu einer Lösung erzeugte Nachbar von der Funktion `pruefe_nachfolgerfunktion` vollständig auf Gültigkeit überprüft wird, obwohl lediglich ein oder zwei Knoten vertauscht worden sind. Ein inkrementeller Ansatz, bei dem nur die veränderten Knoten betrachtet werden, könnte die Performance des Bienenalgorithmus erheblich steigern.

Ausblick und zukünftige Erweiterungen

Das Laubharkprojekt ist nun in einem Zustand, in dem ein Vergleich von Algorithmen relativ einfach möglich ist. Auch das ausführen großer Testmengen ist unbeaufsichtigt möglich, und die Verwaltung der Ergebnisse ist fertig implementiert. In Zukünftigen Ausarbeitungen am Laubharkprojekt könnte man sich der folgenden, wünschenswerten Erweiterungen annehmen:

- Entkopplung der Parameter des genetischen Algorithmus aus ProblemData in ein eigenes "GeneticParameters", wie beim Bienenalgorithmus
- Beschleunigung aller Verfahren durch Kompilierung mit dem MATLAB Coder
- Reproduzierbarkeit der bio-inspirierten Verfahren erlangen, indem Zufallszahlengeneratoren (samt Seeds) zusätzlich zu den Ergebnissen gespeichert werden
- Erweiterung des GUI (Matlab App) auf den aktuellen Projektstand, Integration aller Algorithmen samt aller Parameter zur Einstellung
- Entwicklung eines GUI zur Bedienung des TestManagers
- Dokumentation bisher nicht dokumentierter Funktionen und Klassen

Quellen

[1]: Kruse, Hermann-Josef: *Theoretische Grundlagen zur Implementierung von Lösungsstrategien für das Laubharkproblem*. FH-Bielefeld, Fachbereich Ingenieurwissenschaften und Mathematik (März 2018).

[2]: Kruse, Hermann-Josef: *Theoretische Grundlagen zur allgemeinen Laubharkproblematik*

[3]: Kruse, Hermann-Josef: *Erläuterungen zu den Heuristiken im VBA-Programm*.