WEIHENSTEPHAN · TRIESDORF
University of Applied Sciences

Hochschule Weihenstephan-Triesdorf
Bioprozessinformatik
SS 2022

6. Semester
Agentenbasiertes Modellieren

Exploration

Alina Arneth (Matr.-Nr.: 1368293)
Michael Staab (Matr.-Nr.: 1368264)
Benedikt Traut (Matr.-Nr.: 1380037)
Adrian Wild (Matr.-Nr.: 1366439)

betreut von Niall Palfreyman

Abgabe: 09.07.2022

Gliederung

1. Einleitung

In vielen Bereichen der industriellen Praxis treten heutzutage Optimierungsprobleme auf. Zuverlässige Optimierungsverfahren gewinnen daher zunehmend an Bedeutung. Ein Problem kann dabei meist durch eine Zielfunktion (*Objective Function*) beschrieben werden, welche entweder maximiert oder minimiert werden soll. Ein möglicher Weg, eine solche Funktion zu optimieren sind sogenannte genetische Algorithmen (*GA*). Ein GA ist dabei ein spezieller Algorithmus, welcher dem evolutionären Ansatz der Fortpflanzung von Lebewesen nachempfunden ist.
Der Aufbau eines GAs umfasst einen *Genpool* der enthaltenen *Individuen*. Jedes Individuum besitzt ein Genom bestehend aus *Genen*, welche unterschiedlich ausgeprägt sein können. Bei einem klassischen GA - in diesem Fall der “BasicGA” - gibt es die *Allele* “1” und “0” (*Abb. 1a*).
Der BasicGA nutzt folgende Mechanismen:

- Rekombination: Vereinigung von Strukturen (z.B. Bilden eines Kindgenoms aus zwei Elterngenomen (*Abb. 1b*))
- Mutation: Zufällige Änderungen von Strukturen (z.B. Genom eines Individuums (*Abb. 1c*))

Eine erweiterte Form des klassischen GAs ist der “ExploratoryGA”. Dieser nutzt zusätzlich zu den oben genannten Mechanismen die “Exploration” und besitzt dazu neben den bereits genannten Allelen noch die Ausprägung “?” (*Abb. 1d*). Exploration ist dabei die Fähigkeit einer Struktur, je nach Situation unterschiedliche Ausprägungen anzunehmen. Unser explorativer GA ist dabei dem Algorithmus von Hinton und Nowlan (Hinton, 1987) nachempfunden und soll deren Ergebnisse bestätigen, dass explorative Algorithmen trügerische Funktionen effizienter optimieren als klassische genetische Algorithmen.

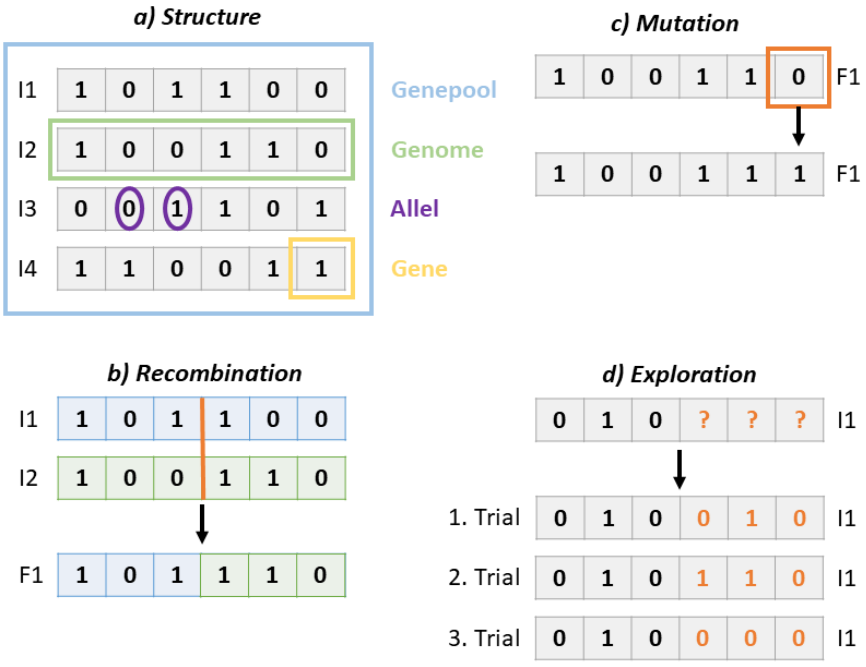

Abb. 1: Struktur des Genpools (*a*) eines GAs und dessen Mechanismen (*b-d*)

Für den Vergleich der beiden oben beschriebenen GAs werden zwei verschiedene Objective Functions benutzt.
Zum einen wurde Watson´s maximal epistatische Objective Function Hierarchical-if-and-only-if (“Mepi”) (Watson, 2007) verwendet. Diese wurde von Watson speziell dafür entwickelt, um die Optimierung mit einfachen GAs so schwierig wie möglich zu machen. Hierbei gibt es zwei Allelkombinationen, die die Funktion minimieren. Diese liegen allerdings so weit wie nur möglich auseinander, wobei sich das nächst schlechtere lokale Minimum in der Mitte befindet (*Abb. 2a*). Somit ist mepi nicht nur schwierig zu optimieren, da es keinen Hinweis gibt, wo sich die beiden globalen Minima befinden, sondern auch trügerisch. Der minimale Wert der Funktion entspricht der Genomlänge.
Zudem wurde eine weitere Objective Function (“Haystack”) implementiert, die die Funktion von Hinton und Nowlan (Hinton, 1987) reproduzieren soll. Anders als bei Mepi gibt es hier nur eine passende Allelkombination, die zum Minimum führt (*Abb. 2b*). Hierbei wurde festgelegt, dass ein durchgängig aus Einsen bestehendes Genom die Funktion mit einem Ergebnis von 0 minimiert. Bei allen anderen Kombinationen ist der Wert immer 1.

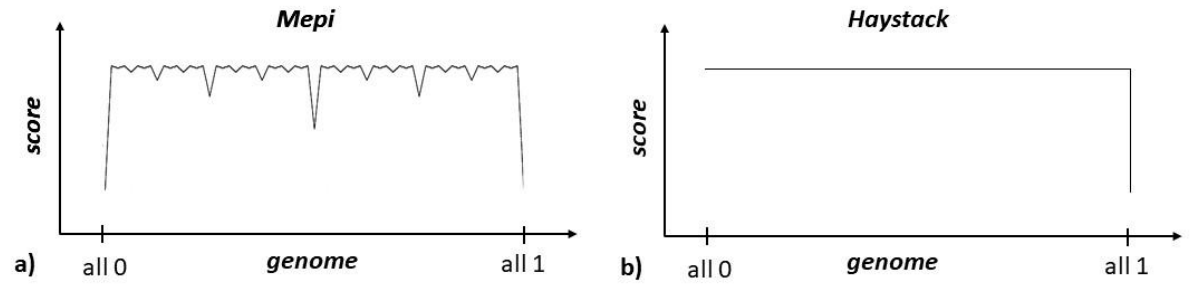

Abb.2: Verlauf der Objective Functions von *(a)* Watson (aus: Watson, 2001) und *(b)* Hinton und Nowlan in Abhängigkeit von den Allel-Ausprägungen im Genom.

2. Manual

2.1 Verwendung

2.1.1 Benötigte Julia Packages

- Statistics.jl
- Agents.jl
- Random.jl
- Plots.jl
- DataFrames.jl
- Dates.jl
- CSV.jl
- TrackingTimers.jl

Damit das Modul funktioniert, müssen diese Pakete über den Julia Paketmanager installiert werden.

2.1.2 Modul einbinden

Um eine Simulation zu starten muss zuerst der Sourcecode des Moduls *GAs* eingebunden und in den globalen Namespace geladen werden:
`include("<Pfad/zum/Projekt>/src/Development/Exploration/GAs.jl")`
`using .GAs`

2.1.3 Wege um das Modul auszuführen

demo()
Mit `demo()` wird ein beispielhafter Simualtionsverlauf mit mehreren *geneticAlgorithms* gestartet. Die Ergebnisse der Simulation werden dann in einem Plot veranschaulicht. Dieser zeigt die besten Werte jedes GAs für jede Generation.

simulate(<geneticAlgorithm>, nSteps)
Die Funktion führt eine einfache Simulation für einen *geneticAlgorithm* aus. Die Ergebnisse werden in einem speziellen Struct zurückgegeben. Siehe `GASimulation`.

compare(<geneticAlgorithms>, nSteps)
Um mehrere *geneticAlgorithms* miteinander zu vergleichen, kann die Funktion `compare(<geneticAlgorithms>, nSteps)` verwendet werden. Dabei ist `<geneticAlgorithms>` ein Vektor von genetischen Algorithmen.
Um die Ergebnisse der einzelnen Simulationen in Tabellen- und Plotform abzuspeichern, kann die Funktion um den Parameter `dirname`, welcher den Pfadnamen für das Exportieren enthält, erweitert werden.
z.B. `compare("C:/Users/<username>/Documents/JULIA",[bga, ega], 1000)`

2.2 Dokumentation

Der Programmcode des Projekts besteht aus dem Modul `GAs` (*GAs.jl*), das auf verschiedene Funktionalitäten aus Julia-Dateien in den Unterordnern `core/`, `mechanisms/` und `utils/` zurückgreift.
Die Simulationen sind implementiert als agentenbasierte Modelle (“ABM”) basierend auf dem Modul *Agents.jl*. ABMs bestehen immer aus einer Menge an Agenten, einem Raum in dem sich Agenten bewegen können und Funktionen, die pro Generation für jeden Agenten aufgerufen werden.

GAs.jl

Das Modul `GAs` stellt Funktionen bereit um Simulationen von genetischen Algorithmen zu initialisieren und durchzuführen.

<genetic-algorithm>_step!()

Die Funktionen `basic_step!()` und `exploratory_step!()` sind die Methoden, die innerhalb der Simulation eines genetischen Algorithmus aufgerufen werden. Dazu

berechnen sie erst eine Matrix, welche den aktuellen Genpool der Population von Agenten (ein Agent pro Zeile) repräsentiert. Diese Matrix dient dann als Eingabe um die Fitness-Scores der Agenten zu berechnen und daraus mittels Tournament-Selection sowie Mutation eine neue Generation an Agenten zu erzeugen.

initialize()

Die Funktion `initialize()` besitzt jeweils eine Methode für sowohl `BasicGA`, als auch `ExploratoryGA`.
Die Funktionen erzeugen, basierend auf einer Instanz des jeweiligen Algorithmen-structs das ABM mit den entsprechenden Parametern und füllen es mit Agenten des entsprechenden Typs.

simulate()

Die Funktion simulate() besteht aus zwei Methoden, je eine für den `BasicGA` und den `ExploratoryGA`.
Die Funktionen initialisieren das entsprechende ABM (siehe `initialize()`), definieren, welche Daten den ganzen Simulationsverlauf über gesammelt werden sollen (adata), führen die Simulation durch und bereiten die Daten für eine einfachere Analyse und bessere Vergleichbarkeit zwischen den Algorithmen auf.

compare()

Es existieren zwei Methoden der Funktion `compare()`.

compare(geneticAlgorithms::Vector{T}, nSteps=100; [...])

Diese `compare()`-Methode berechnet wie viele Genom-Evaluierungen pro Generation (“Step”) stattfinden und ermittelt daraus für alle Algorithmen die Anzahl an Generationen pro Simulation um insgesamt gleich viele Genom-Evaluierungen durchzuführen. Danach startet die Funktion die entsprechenden Simulationen für alle übergebenen Algorithmen und Parameter.

Nachverfolgung der Laufzeit

Um die Laufzeit der Simulationen zu messen, wird das Modul *TrackingTimers.jl verwendet*. Der Vorteil des Moduls ist es, dass damit die Laufzeiten in ein *DataFrame* gespeichert werden können. Somit ist es möglich, die Daten als CSV Datei zu exportieren und später zu analysieren.
Initialisierung: `runtimes = TrackingTimer()`
Verwendung: `TrackingTimers.@timeit runtimes ”<name>” <simulation>`
Für mehr informationen: https://github.com/ericphanson/TrackingTimers.jl

Multi-Threading

Beide `compare()`-Methoden bieten einen optionalen Parameter `multiThreading`. Dieser wirkt sich effektiv jedoch nur auf die erste `compare()`-Methode aus.
Ist dieser Parameter gleich `true` (und ist Julia mit mehr als einem einzelnen Thread gestartet), so werden mithilfe des Moduls *Base.Threads* die einzelnen Simulationen auf je einen Thread aufgeteilt um eine verbesserte Gesamtlaufzeit und CPU-Auslastung zu

erreichen. Hierbei muss jedoch manuell auf die Integrität der Daten geachtet werden, weshalb hier mit sogenannten "`Locks`" gearbeitet wird um Threads daran zu hindern gleichzeitig auf dasselbe Objekt zuzugreifen. Ob tatsächlich Multi-Threading genutzt wird ist daran ersichtlich, dass den Konsolenausgaben "`[Thread n]`" hinzugefügt wird, wobei *n* die Thread ID ist.

(Hinweis: Julia startet standardmäßig mit nur einem Thread. Um Julia mit mehr Threads zu starten muss die Anzahl an Threads beim Start spezifiziert werden:
`julia -t <numThreads>`)

compare(dirname::String, geneticAlgorithms::Vector{T}, nSteps=100; [...])

Diese `compare()`-Methode erweitert die obige `compare()`-Methode um die Speicherung und Visualisierung der Simulationsdaten. Dazu führt sie die Simulationen mittels der anderen `compare()`-Methode durch, erstellt die Diagramme und speichert diese schließlich gemeinsam mit den gesammelten Daten im .csv-Format im angegeben Verzeichnis *dirname* ab.

demo()

Die Funktion `demo()` ist für eine einfache Präsentation des Moduls erstellt worden. Man benötigt keine Parameter um sie auszuführen und sie wird exportiert. Die Funktion erstellt mehrere GAs, lässt die Simulation laufen und zeigt das Ergebnis als Plot an.

core/

Der Ordner `core` enthält die für die Simulationen notwendigen Datenstrukturen und Funktionalitäten. In diesem Ordner sind die verschiedenen Algorithmen sowie algorithmenspezifische Agenten und Allele definiert.

agents.jl

In der Datei *agents.jl* werden die algorithmenspezifischen Agenten definiert. Jeder Agent besitzt dabei, zusätzlich zu den seitens Agents.jl benötigten Attributen, ein Genom - einen Vektor aus Genen - sowie ein Feld Score, das den Wert der Objective Function, evaluiert am momentanen Genom, enthält.

BasicGAAgent (struct)

siehe *agents.jl*. Der `BasicGAAgent` besitzt ein Genom, das aus Genen mit den möglichen Allelen `BasicGAAlleles` besteht.

ExploratoryGAAgent (struct)

siehe *agents.jl*. Der `ExploratoryGAAgent` besitzt ein Genom, das aus Genen mit den möglichen Allelen `ExploratoryGAAlleles` besteht.

algorithms.jl

Die Datei *algorithms.jl* definiert structs zur Spezifizierung von genetischen Algorithmen und stellt entsprechende Konstruktoren bereit.

GeneticAlgorithm (type)

Der Typ `GeneticAlgorithm` definiert einen Supertyp für alle genetischen Algorithmen.

BasicGA (struct)

Instanzen vom struct `BasicGA` besitzen alle notwendigen Attribute um Simulationen mit dem BasicGA-Algorithmus durchzuführen.
Dazu besitzen sie:

- *`nIndividuals`*
  Die Anzahl an Individuen in der Population
- *`nGenes`*
  Die Genomlänge eines jeden Individuums
- *`mu`*
  Gibt an, welcher Anteil des gesamten Genpools der Population pro Generation mutiert wird.
- *`useHaystack`*
  Spezifiziert, ob Hinton und Nowlan's Haystack oder Watson's Mepi als Objective Function verwendet werden soll. Default: *false* (Mepi)
- *`M`*
  Die Größe des Raumes in dem sich die Agenten bewegen (intern benötigt von *Agents.jl*)

Zudem stellt das struct einen entsprechenden Konstruktor zur Instanziierung eines `BasicGA` bereit.

ExploratoryGA (struct)

Instanzen vom struct `ExploratoryGA` besitzen alle notwendigen Attribute um Simulationen mit dem ExploratoryGA-Algorithmus durchzuführen.
Dazu besitzen sie (siehe auch `BasicGA`):

- *`nIndividuals`*
- *`nGenes`*
- *`mu`*
- *`useHaystack`*
- *`nTrials`*
  Anzahl an explorativen Suchen, die ein Agent pro Generation durchführt, bevor weiter zur Selektion gegangen wird.
- *`M`*

Zudem stellt das struct einen entsprechenden Konstruktor zur Instanziierung eines `ExploratoryGA` bereit.

Helper Funktionen

paramstring()

Diese Funktion gibt für einen übergebenen Algorithmus einen beschreibenden "Parameter-String" zurück. Dieser enthält Informationen zur Art des GAs und allen relevanten Parametern (`<<GA-Name>>--<<param1>>-<<value1>>--<<param2>>-<<value2>>[…]`).

Diese Funktion wird beispielsweise zur Benennung von gespeicherten Simulationsergebnissen verwendet.

pointToUnderscore()

Diese Funktion wandelt innerhalb eines Strings alle "`.`" in "`_`" um.

alleles.jl

Die Allele bezeichnen mögliche Ausprägungen eines jeden Gens. Diese unterscheiden sich je nach Algorithmus.

BasicGAAlleles (enum)

Beim BasicGA kann ein Gen entweder gesetzt (*`bOne`*) oder nicht gesetzt (*`bZero`*) sein.

ExploratoryGAAlleles (enum)

Beim ExploratoryGA kann ein Gen zusätzlich zu den Möglichkeiten gesetzt (*`eOne`*) und nicht gesetzt (*`eZero`*) noch unbestimmt (*`qMark`*) sein. Unbestimmte Gene werden im ExploratoryGA bei der Genom-Evaluierung zufällig gesetzt. (siehe `plasticity()`)

mechanisms/

Dieser Ordner beinhaltet Programmcode, der die Funktionalitäten genetischer Algorithmen sowie explorativer genetischer Algorithmen realisiert. Er beinhaltet beispielsweise Funktionen zu Mutation, Selektion und Rekombination.

encounter.jl

In der Datei *encounter.jl* liegen alle Funktionen zur Ermittlung der Eltern für die nächste Generation. Derzeit ist nur die Tournament Selektion implementiert.

encounter()

Mit dem Tournament Selektion Verfahren werden die Eltern für die nächste Generation ausgewählt. Dabei werden immer 2 zufällige Individuen miteinander verglichen. Das Individuum mit dem größeren Fitness Wert wird als Elternteil ausgewählt.

fitness.jl

In der Datei *fitness.jl* wird die Fitness der Individuen basierend auf der jeweiligen Objective Function berechnet.

mepi()

Das Ergebnis von Watson's maximal epistatischen Objective Function (Watson, 2007) für ein Genom wird mit `mepi()` berechnet.

haystack()

Das Ergebnis von Hinton und Nowlan's Haystack Funktion (Hinton, 1987) für ein Genom wird mit `haystack()` berechnet.

fitness()

Die Funktion `fitness()` besitzt zwei Methoden. Die Basismethode berechnet die normalisierten Fitnesswerte der Individuen auf Basis der Ergebnisse der *Objective Function*. Für einen `ExploratoryGA` wird diese für jeden Lernversuch erneut aufgerufen und schlussendlich der beste Fitnesswert und das zugrunde liegende Ergebnis der Objective Function zurückgegeben.

mutate.jl

mutate!()

Die Funktion `mutate!()` mutiert eine gegebene Populations-Matrix (*`genpool`*) mittels gegebener Mutationsrate *`mu`.* Dabei wird jedes einzelne Matrixelement - also jedes einzelne Gen - mit Wahrscheinlichkeit *`mu`* durch ein zufälliges Allel ersetzt. Die Funktion arbeitet dabei direkt auf der übergebenen Matrix.
Die Basisimplementierung erwartet eine Matrix von Integer-werten als *`genpool`* sowie einen Vektor *`alleles`*, der die Menge der möglichen Allele - jeweils ein Integer pro möglichem Allel - enthält. Die Methode wählt dann mittels des übergebenen *`Casinos`* zufällige Loci im Genpool aus, welchen dann jeweils ein zufälliges Allel aus dem Vektor *`alleles`* zugewiesen wird.
Die Enumimplementierung erwartet eine Matrix von Enums als *`genpool`* und ermittelt die Menge der möglichen Allele als `instances(enumtype)` des Typs der Matrixelmente. Die Mutation des Genpools erfolgt dann durch das Umwandeln der Enums in Integer und dem Aufruf der Basisimplementierung mit der umgewandelten Matrix. Schlussendlich wird die mutierte Matrix wiederum in ihre Enum-Form transformiert und über Broadcast dem ursprünglich übergebenen *`genpool`* Parameter zugewiesen.

plasticity.jl

plasticity()

Diese Funktion implementiert die explorative Suche eines `ExploratoryGA`. Dazu wird aus dem *`genpool`* (ein Agent pro Zeile) eines ExploratoryGAs mit unbestimmten (`qMark`) Genen eine binäre Matrix erstellt. Dabei wird jedes `qMark` mit 50% als gesetzt (`eOne`) oder nicht gesetzt (`eZero`) interpretiert. Für die Generierung von Zufallszahlen wird die übergebene *Casinos.jl* Instanz verwendet.

recombine.jl

recombine()

Die Funktion `recombine()` erzeugt aus den Genomen der Gewinner der Selektion (“Elternteile”) eine neue Generation (“Kinder”). Dazu wird je das Genom zweier Gewinner an einem zufälligen Crossing-Over Punkt geteilt und versetzt zusammengefügt. Aus zwei Elternteilen entsteht somit ein Kind, wobei der erste Teil des Genoms von Elternteil 1 übernommen wird und der zweiten Teil des Genoms von Elternteil 2 (*Abb. 1b*).

Die Methode benötigt dazu den `genpool`. Da ein (1) Kind aus zwei Elternteilen entsteht, erwartet die Methode einen Vektor von 2*N Indizes (N ist Populationsgröße). Diese Indizes referenzieren je einen Selektions-Gewinner.

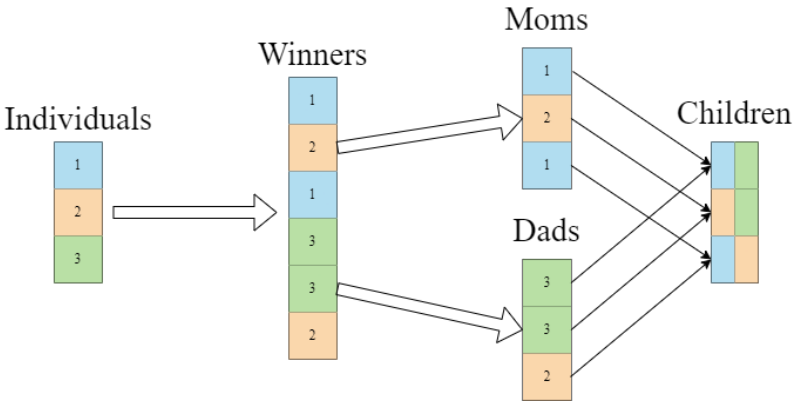

Abb.3: Konstruktion von Kindern durch Rekombination aus Elter-Individuen

Die Gewinner der Selektion werden hier in zwei Hälften ("`moms`", "`dads`") aufgeteilt und der erste mit dem Zweiten kombiniert. Das erste "Kind" ist somit das Kind von Individuum 1 und Individuum N+1, Kind 2 ist von Individuum 2 und N+2, usw.
Um die Rekombinierungen durchzuführen wird erst eine logische Matrix erzeugt, welche basierend auf den Crossing-Over Punkten darstellt, welcher Teil des Genoms des jeweiligen Elternteils in den Kind-Organismus übernommen wird. An Stellen mit einer 1 soll das entsprechende Gen der *Mom*, an Stellen mit einer 0 das des *Dads* verwendet werden.

1	1	1	1	1	0
1	1	1	0	0	0
1	1	1	1	0	0

Abb.4: beispielhafte ParentsLogicalMatrix

Im nächsten Schritt wird eine Matrix berechnet, welche für jedes Gen einen Index enthält, mit dem auf das entsprechende Gen im Elternteil referenziert werden kann. Das erste Gen des ersten Individuums der Ursprungsgeneration entspricht dem Index 1, das zweite Gen des ersten Individuums dem Index 2. Nimmt man eine Genomlänge von 6 Genen an, so entspricht das erste Gen des zweiten Individuums dem Index $nGenes + 1 = 6 + 1 = 7$, usw. Allgemein kann man das n-te Gen des m-ten Individuums also mit dem Index $((m - 1) \cdot nGenes) + n$ indexieren.
Die Berechnung dieser Indexe findet im Code in folgender Zeile statt:

```
indices = ((parentsLogicalMatrix .* moms .+ .!parentsLogicalMatrix .* dads)) .* nGenes  .+ collect(1:nGenes)'
```

und soll im Folgenden genauer betrachtet werden.
Als Vorbereitung wird der Vektor der Gewinner in den zwei vorherigen Operationen in `moms` und `dads` aufgeteilt und jedes Element um 1 verringert um den korrekten "Offset" ($(m - 1)$ * $nGenes$) berechnen zu können.
Der erste Klammerausdruck

```
(parentsLogicalMatrix .* moms .+ .!parentsLogicalMatrix .* dads)
```

berechnet den Vorfaktor des Offsets $(m - 1)$ für jedes Gen der "Kind-Generation" zum Indizieren. Im nächsten Schritt

```
([...]) .* nGenes
```

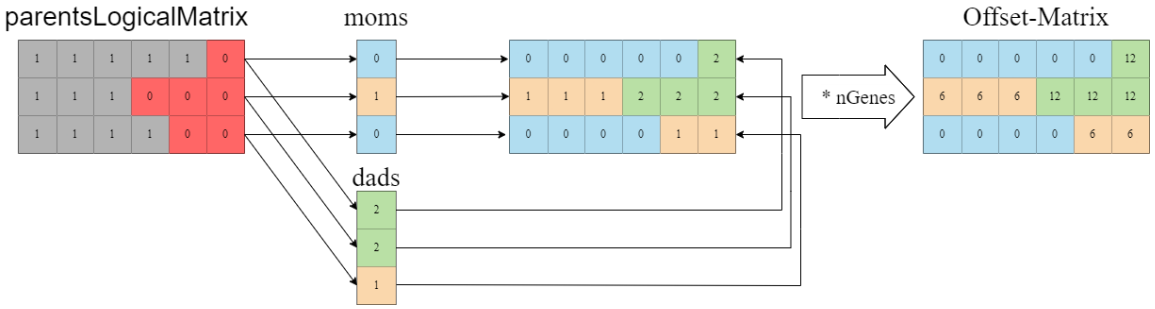

Abb.5: Berechnung des Offsets aus ParentsLogicalMatrix und den SelectionWinners

wird die Berechnung des Vorfaktors $(m - 1) \cdot nGenes$ abgeschlossen (*Abb. 5*). Anschließend wird jeder Spalte die entsprechenden Spaltennummer hinzuaddiert:

```
indices = [...] .+ collect(1:nGenes)'
```

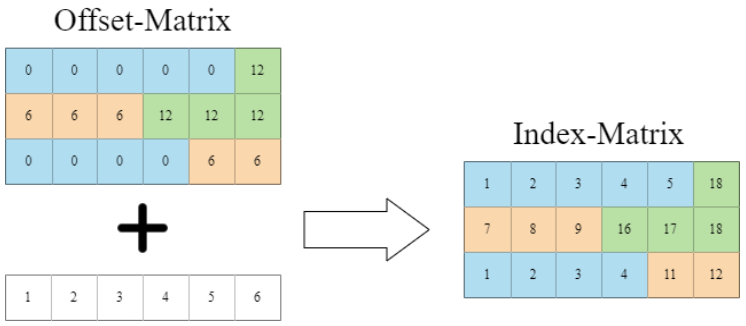

Abb.6: Berechnung der Indices aus Offset und Gen-Nummer

Um mit der berechneten Index-Matrix die Gene richtig referenzieren zu können, muss die Matrix der Ursprungsgeneration jedoch noch transponiert (siehe auch *transpose.jl*) werden, so ergeben sich die letzten Zeilen der Funktion:

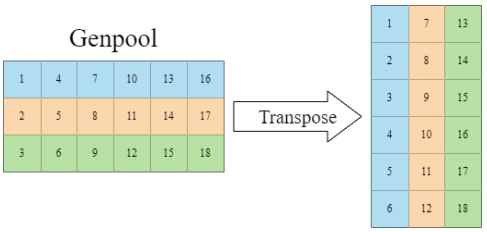

Abb.7: Visualisierung der Transponierung des genpools

```
newGenpool = transpose(genpool)[indices]
return newGenpool
```

utils/

Der Ordner *utils/* enthält Julia-Dateien mit Funktionalitäten, die für das Verarbeiten, Anzeigen und Speichern von Ergebnissen genutzt werden, sowie weitere Hilfswerkzeuge für die Durchführung von Simulationen.

display.jl

Die Datei *display.jl/* stellt Funktionen bereit, welche dem Benutzer die Ergebnisse einer Simulation in Form eines Plots anzeigen.

displayCompareMinimumScoresPlot()

Die Funktion verarbeitet die Ergebnisse einer Simulation und erstellt einen Plot, welcher die besten Werte jedes GAs zu jedem Step zeigt. Dieser Plot wird anschließend dem Benutzer angezeigt.

results.jl

Die Datei *results.jl* stellt structs zum Speichern der Simulationsergebnisse bereit. Für eine einzelne Simulation wird das struct `GASimulation` definiert, welches die Endzeit der Simulation (*`timestamp`*), den ausgewählten *`GeneticAlgorithm`* (*`algorithm`*) und die während der Simulation gesammelten Daten (*`simulationDF`*) enthält. Des Weiteren wird hier das struct `GAComparsion` definiert, das die zu vergleichenden Simulationen (*`simulations`*) und neben der Endzeit (*`timestamp`*) auch einen `DataFrame` mit den einzelnen gemessenen Simulations-Laufzeiten und jeweils verwendeten Thread-ID's (*`runtimes`*) enthält.

plotting.jl

In *plotting.jl* sind verschiedene Funktionen zur Visualisierung der Simulationsergebnisse implementiert.

finalizePlot!()

Die Funktion `finalizePlot!()` sorgt für eine gute Anzeigequalität und wird am Ende jeder Plot-Funktion aufgerufen, wobei der entsprechende Plot übergeben wird.

compareMinimumScore()

Zum Vergleich verschiedener Simulationen mit unterschiedlichen Parametern wird mit `compareMinimumScore()` für jeden `GeneticAlgorithm` der minimale Wert der ausgewählten Objective Function über alle Individuen pro Anzahl an Genom-Evaluierungen angezeigt.

scoreOverTime()

Für einen genauen Verlauf des Wertes aller Individuen steht die Funktion `scoreOverTime()` zur Verfügung. Hier wird der Wert der Objective Function für jedes einzelne Individuum pro Anzahl an Genom-Evaluierungen visualisiert. Da der Plot allerdings sehr rechenintensiv ist, wird er nicht mit den Simulationsergebnissen ausgegeben.

scoreSpanOverTime()

Durch die Funktion `scoreSpanOverTime()`, welche jeweils das Minimum, Maximum und den Mittelwert der Objective Function pro Anzahl an Genom-Evaluierungen plottet, kann man trotzdem einen guten Überblick über die Verteilung der erzielten Werte in der Gesamtpopulation erhalten.

topTierOverTime()

Dies wird auch durch das mit `topTierOverTime()` erzeugte Histogram unterstützt. Der Plot zeigt die Anzahl der Individuen pro Anzahl an Genom-Evaluierungen an, die in einem vorher definierten prozentualen Bereich um den besten Wert liegen.

allelicExpressionNumber()

Die Verteilung der Allele über den Simulationsverlauf hinweg wird mit Hilfe der Funktion `allelicExpressionNumber()` visualisiert. Hier wird zu jedem Zeitpunkt der Mittelwert der Anzahl der verschiedenen Gen-Ausprägungen im Genom aller Individuen angezeigt.

save.jl

Die Datei *save.jl* stellt Funktionen zum Speichern der erzeugten Plots und gesammelten Daten bereit.

processSimulationData()

Da viele Plots eine ähnliche Verarbeitung der Daten benötigen, wurde die Berechnung des Minimums, Maximums und des Mittelwerts der Objective Function pro Anzahl an Genom-Evaluierungen indie Funktion `processSimulationData()` ausgelagert.

generateSimulationPlots()

Erzeugt die zu speichernden Plots für eine Simulation (`GASimulation`) und gibt diese in Form eines Dictionaries zurück.

formatTimeStamp()

Die Funktion `formatTimeStamp()` formatiert die Endzeit der Simulationen in ein geeignetes Format, sodass auch der Sekundenwert angezeigt wird. Dies wird für den Dateinamen der gespeicherten Plots und Simulationsdaten verwendet.

savePlots()

Mit der Funktion `savePlots()` können die visualisierten Ergebnisse schließlich exportiert werden. Die Funktion besitzt insgesamt drei Methoden. Bei einer einzelnen Simulationen (`GASimulation`) umfasst dies die zuvor in `generateSimulationPlots()` erzeugten Plots. Diese werden für eine `GAComparsion` um den das Vergleichsplot aus `compareMinimumScore()` erweitert. Das Abspeichern der allgemeinen Simulation-Plots bleibt hierbei allerdings optional, wird aber per default ausgeführt. Die Plots werden schließlich in png- und pdf-Format mit einem generierten Dateinamen heruntergeladen.

saveData()

Die während den Simulationen gesammelten Daten können mit Hilfe von `saveData()` gespeichert werden. Hierbei stehen wieder drei Methoden zur Verfügung. Für eine

`GASimulation` werden die Daten der Individuen über der Zeit gespeichert. Dies wird bei einer `GAComparsion` um die Daten der gemessenen Laufzeiten der einzelnen Simulationen erweitert. Die Datensätze werden im csv-Format exportiert.

casinos.jl

Für das Erzeugen von Zufallszahlen wird das von Niall Palfreyman verfasste Modul *Casinos.jl* verwendet.

transpose.jl

transpose()

In *transpose.jl* wird das Julia-Modul Base um zwei weitere Methoden für `transpose()` erweitert, um diese sowohl für eine Matrix als auch einen Vektor bestehend aus *Enums* aufrufen zu können.

3. Motivation

"phenotypic accommodation can precede, rather than follow, genetic change, in adaptive evolution" (EES, 2022)

Die Extended Evolutionary Synthesis (*EES*) besagt unter anderem, dass (phenotypische) Evolution nicht allein durch genetische Mutation und Rekombination getrieben ist, sondern, dass explorative Suche während der Lebenszeit ebenfalls eine sehr wichtige Rolle spielt. Denn da es sehr unwahrscheinlich ist, dass reine genetische Suche komplexe Organismen, wie zum Beispiel Menschen, hervorgebracht hat, liegt es nahe, dass weitere Mechanismen, wie Exploration eine entscheidende Rolle spielen. Dabei steht außer Frage, dass die evolutionäre Entwicklung von komplexen Organismen als eines der schwierigsten Optimierungsprobleme angesehen werden kann. Ein klassischer genetischer Algorithmus, welcher ein Werkzeug zur theoretischen Betrachtung von Evolution ist, sollte daher durch die Erweiterung um den Mechanismus der Exploration komplexe Probleme zunehmend besser lösen können.

3.1 Hypothese

Erweitert man einen genetischen Algorithmus um explorative Suche, so ist es diesem ExploratoryGA möglich komplexere Probleme zu minimieren.

3.2 Null-Hypothese

Es gibt einen genetischen Algorithmus, der ohne die Zuhilfenahme explorativer Suche eine Objective function minimiert, welche der ExploratoryGA bei gleicher Komplexität nicht minimieren kann.

3.3 Vorgehen

Da der ExploratoryGA durch seine Lernversuche mehr Genom-Evaluierungen als der BasicGA durchläuft, wurde eine Methode gesucht um die beiden besser vergleichen zu können. Deshalb wird die Generations-Anzahl des BasicGA mit der Lernversuchs-Anzahl des ExploratoryGAs multipliziert. So durchlaufen beide die selbe Anzahl an Genom-Evaluierungen.
Um herauszufinden, ob der ExploratoryGA eine Objective Function besser als der BasicGA minimieren kann, sollte erst einmal überprüft werden ab welcher Komplexitätsstufe einer der beiden GAs die Objective Function nicht mehr minimieren kann. Für einen generellen Überblick wurden zunächst breit gefächerte Simulationen mit verschiedenen Parametervariationen ausprobiert. So wurden unterschiedliche Mutationsraten, Genomlängen und Individuen-, Generations- und gegebenenfalls Lernversuch-Anzahlen miteinander kombiniert.
Schließlich wurden alle Parameter außer der Genomlänge festgesetzt. Hier wurden die Individuen-, Generations- und Lernversuch-Anzahlen jeweils auf 1000 und die Mutationsrate auf durchschnittlich zwei Genmutationen pro Generation festgelegt.
Die Genomlänge dient hierbei somit als einfacher Regler der Komplexität: Je länger die Genomlänge, desto schwieriger ist es, das globale Minimum zu finden.
Um den Grenzpunkt zu finden, wurden erneut Simulationen mit verschiedensten Genomlängen laufen gelassen.
Sobald der Grenzpunkt für einen der GAs gefunden wurde, wurden mehrere Simulationen mit ähnlicher Genomlänge durchgeführt, um den Bereich um diese Stelle genauer zu untersuchen.
Außerdem wurde die Komplexität weiterhin erhöht, um auch den Grenzpunkt des anderen GAs zu ermitteln.
Die dadurch ermittelten Simulationen wurden für die Auswertung verwendet.

4. Ergebnisse

Da der BasicGA und der ExploratoryGA im Bezug auf zwei verschiedene Objective Functions betrachtet wurden, werden diese hier gesondert betrachtet.

4.1 Haystack Funktion

Der erste Punkt, an dem ein BasicGA die Haystack Funktion nicht mehr minimieren konnte, liegt bei einer Genomlänge von 29 (Abb. 8a). Bei dieser Genomlänge hat der ExploratoryGA das Minimum jedoch noch finden können (Abb. 8b). Erst bei einer Genomlänge von 32 findet auch der ExploratoryGA das globale Minimum nicht mehr (Abb. 8d).

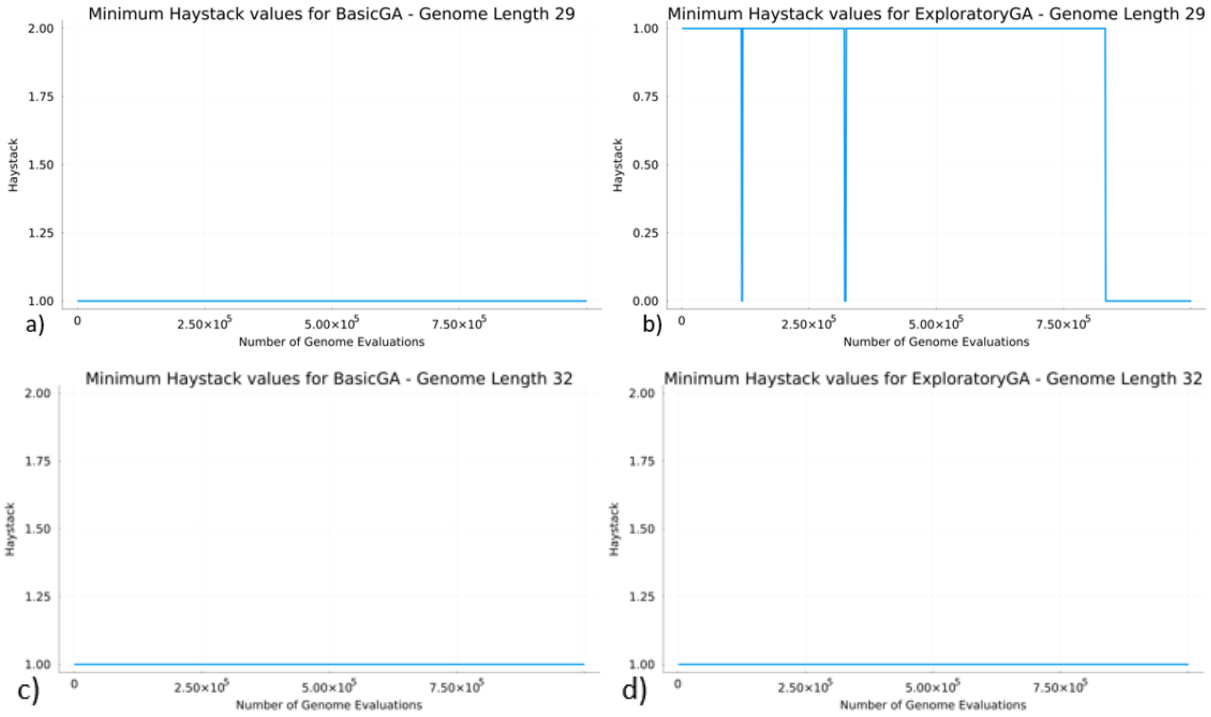

Abb.8: Minimum des Haystack-Werts zu jeder Genom-Evaluierungsanzahl des BasicGAs bei einer Genomlänge von (*a*) 29 und (*c*) 32 und des ExploratoryGAs bei einer Genomlänge von (*b*) 29 und (*d*) 32

Betrachtet man die Verteilung der Allele der beiden Algorithmen über die Zeit, so wird deutlich, dass der BasicGA rein zufällig sucht und damit nicht zum Minimum findet (Abb. 9). Der ExploratoryGA sucht ebenfalls zufällig, im Gegensatz zum BasicGA jedoch findet er nach circa 350 Generationen per explorativer Suche zufällig das Minimum. In den folgenden Generationen manifestiert sich diese optimale Genkombination fortlaufend, was an der fallenden Anzahl an “0en” und darauffolgend auch “?” sichtbar ist (Abb. 10).

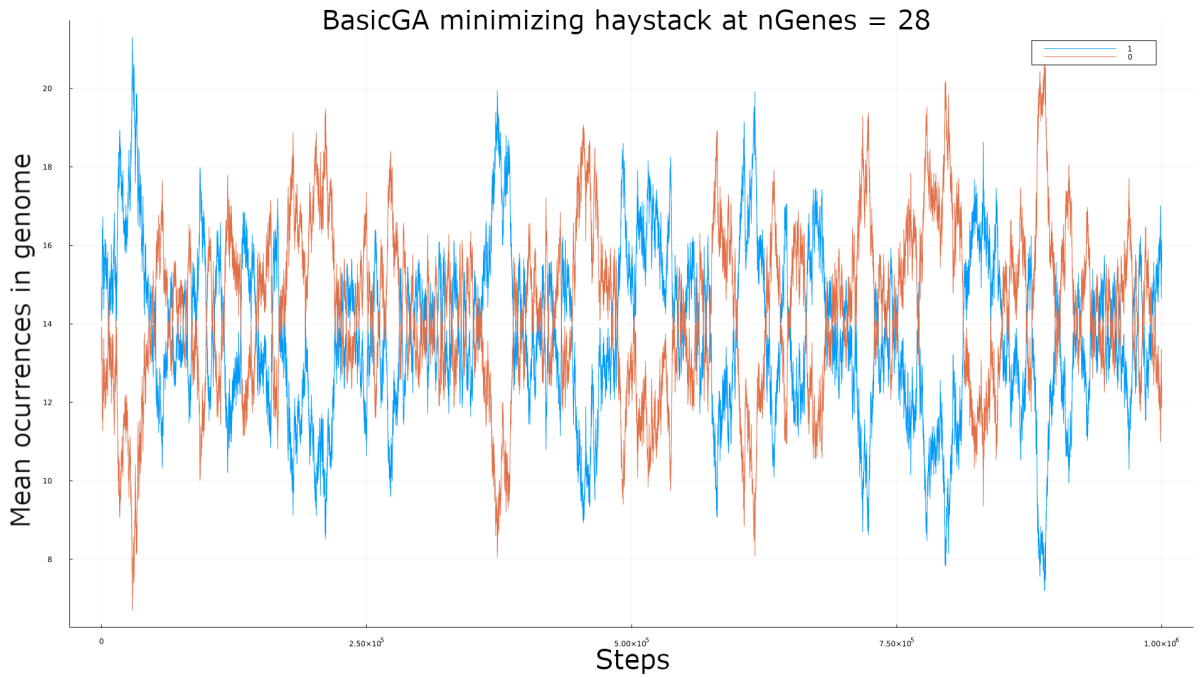

Abb. 9: Durchschnitt der Allelanzahlen eines BasicGAs mit einer Genomlänge von 28

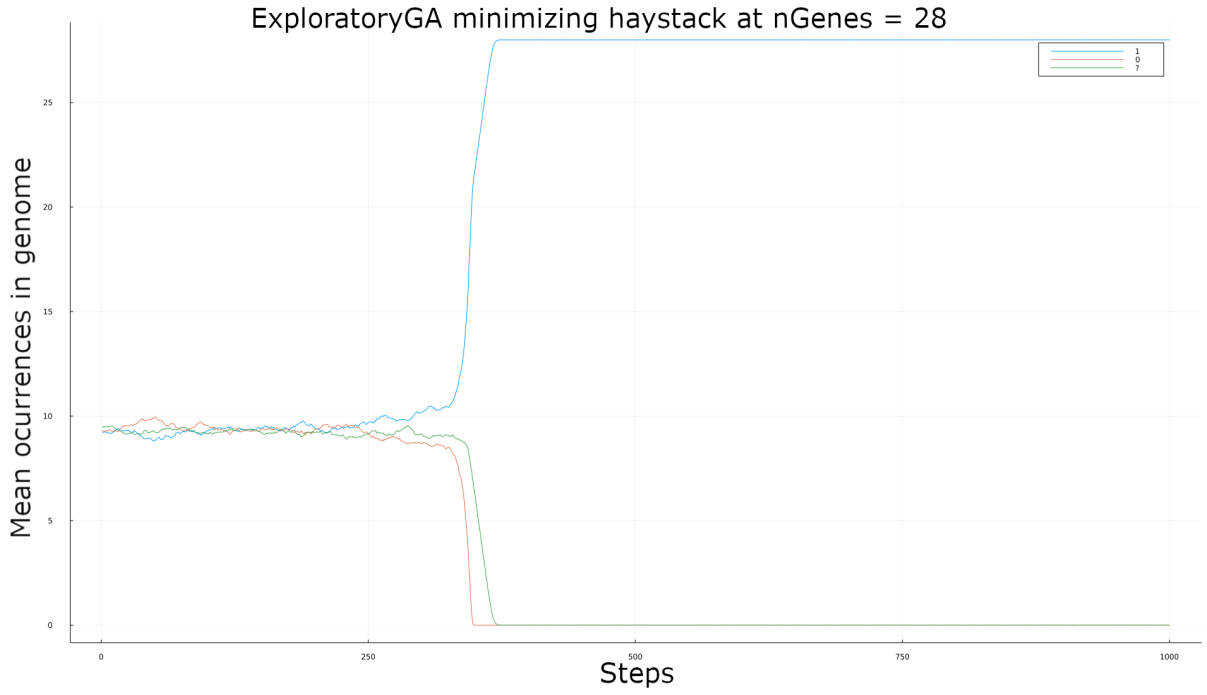

Abb. 10: Durchschnitt der Allelanzahlen eines ExploratoryGAs mit einer Genomlänge von 28

4.2 Mepi Funktion

Der BasicGA konnte in den beobachteten Simulationen die Mepi Funktion zuletzt bei einer Genomlänge von 80 minimieren. Im Gegensatz dazu konnte der letzte Zeitpunkt, an dem der ExploratoryGA das globale Minimum gefunden hat, bei 100 beobachtet werden (*Abb. 11b*). Das Ergebnis der Objective Function ist dabei allerdings abhängig von dem jeweiligen Simulationslauf selbst. So wird nicht immer das globale Minimum gefunden, sondern der ExploratoryGA stagniert in einem Lokalen wie der BasicGA (*Abb. 11b*). Hierbei unterscheiden sich zudem die gefundenen lokalen Minima und so liegt der Wert des ExploratoryGAs bei manchen Simulationen unterhalb dem des BasicGAs oder andersrum.

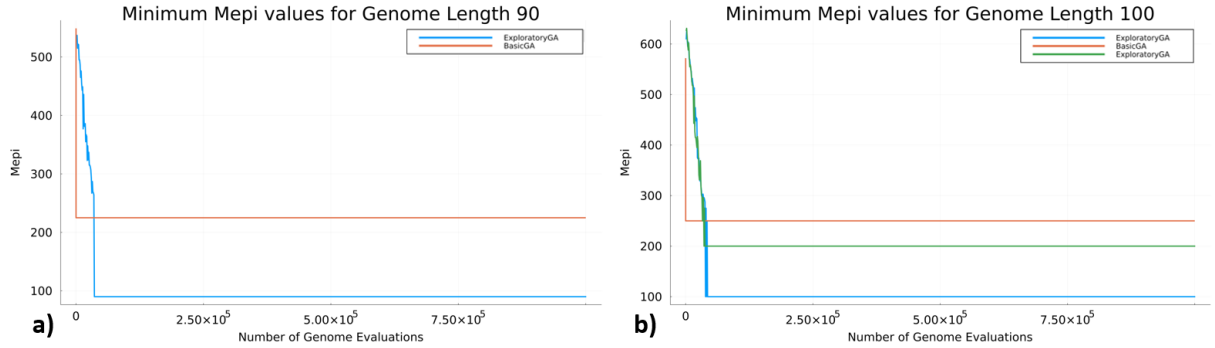

Abb.11: Minimum des Mepi-Werts zu jeder Genom-Evaluierungsanzahl der GAs bei einer Genomlänge von (*a*) 90 und (*b*) 100

Bei der Betrachtung der durchschnittlichen Allelverteilung der Individuen über die Zeit ist anzumerken, dass beim ExploratoryGA die “?” Gen-Ausprägungen in den beobachteten Simulationen zu Beginn relativ stark abfallen und spätestens nach ungefähr 250 Generationen in keinem Genom der Individuen mehr enthalten sind. Dies ist unabhängig davon ob es sich um ein lokales oder globales Minimum handelt (*Abb. 12, 13*). Der Verlauf der “1en” und “0en” gleicht sich zu diesem Zeitpunkt ebenfalls aus, unabhängig von der Art des Minimums. Bei einer Genomlänge von 120 wurde von dem ExploratoryGA zwar nicht das globale, dafür aber das nächst schlechtere Minimum erzielt. Dieses wird erreicht, wenn die erste Hälfte des Genoms nur aus “0” oder nur “1” besteht und die Zweite aus dem Gegenteil (*Abb. 12*).
Anders als bei der Haystack Funktion fluktuiert die Verteilung der Allele des BasicGA über die Zeit nicht sondern stagniert (*Abb. 12, 13*). Dies spiegelt das Stagnieren zum frühen Zeitpunkt im lokalen Minimum wieder.

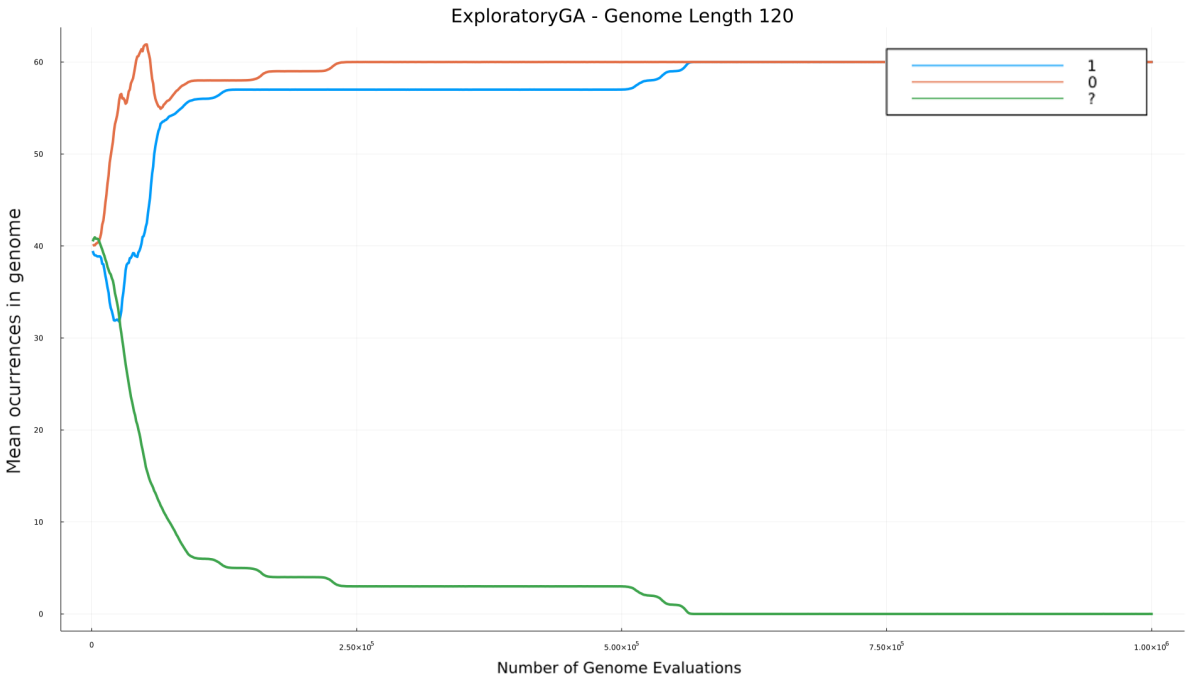

Abb. 12: Durchschnitt der Allelanzahlen eines ExploratoryGAs mit einer Genomlänge von 120

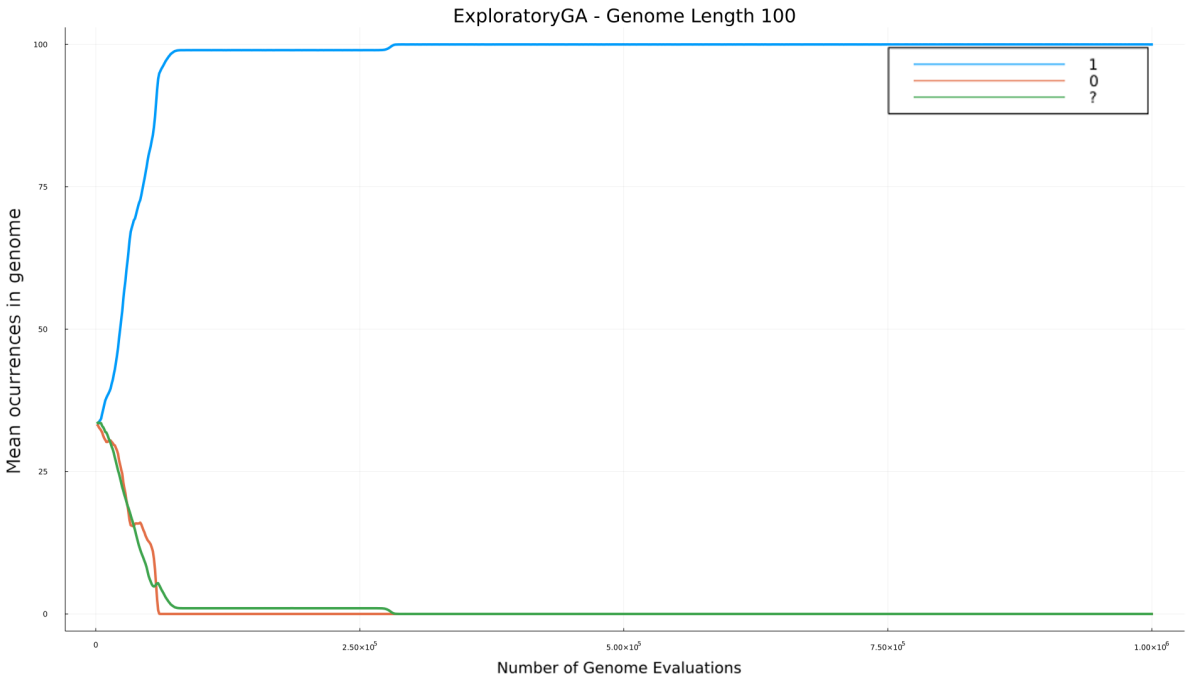

Abb. 13: Durchschnitt der Allelanzahlen eines ExploratoryGAs mit einer Genomlänge von 100

4.3 Laufzeiten

Der ExploratoryGA ist sowohl für die Haystack Funktion, als auch für die Mepi Funktion deutlich schneller als der BasicGA.

Tabelle.1: Laufzeiten der beiden genetischen Algorithmen exemplarisch bei einer Genomlänge von 32

	Haystack	*Mepi*
BasicGA	53 min	134 min
ExploratoryGA	9 min	90 min

5. Diskussion

Mit diesen Ergebnissen wird gezeigt, dass explorative Suche vorteilhaft ist. Bei beiden Objective Functions schlägt sich der Algorithmus, welcher die explorative Suche implementiert, besser. Dies zeigt sich zum Einen beim Erreichen des globalen Minimums. Auch wenn für den ExploratoryGA die Wahrscheinlichkeit größer ist die Funktion zu minimieren, hängt dies von der jeweiligen zufälligen Initialisierung der Simulation selbst ab. So kann es sein, dass der Exploratory in verschiedenen Vergleich-Simulationen ein besseres lokales Minimum findet, allerdings auch andersherum.
Durch das Abfallen und vollständige Verschwinden des “?” Allels im Genom der Individuen lässt sich erkennen, dass sich günstige Allelkombinationen dauerhaft durchsetzen.
Der Vorteil eines ExploratoryGAs zeigt sich auch in der Dauer der Laufzeit.
Das liegt daran, dass der BasicGA deutlich mehr Matrix- und Vektor Zugriffe für die Simulation benötigt. Der Unterschied fällt bei der Mepi Funktion kleiner aus, da die Funktion an sich sehr rechenintensiv ist und somit die zusätzlichen Matrix- und Vektor Zugriffe beim BasicGA nicht mehr so stark ins Gewicht fallen.
Auch wenn durch das Anpassen der Genom-Evaluierung die beiden GAs vergleichbarer gemacht wurden, durchläuft der BasicGA dadurch öfter Rekombinationen und Mutationen als der ExploratoryGA.
Auf Grundlage der erzeugten Daten können wir die Nullhypothese somit ablehnen.

Da der Erfolg der Algorithmen stark von der zufälligen Initialisierung der Individuen abhängig ist, sind die erzeugten Daten jedoch nur bedingt aussagekräftig. Um hier eine aussagekräftigere Datenlage zu erhalten, sollten die betrachteten Simulationen noch öfter durchgeführt werden, um den Einfluss der Initialisierung im Mittel auszugleichen.
Zusätzlich zeigte sich in den “explorativen”, breit gefächerten Simulationen zu Beginn (vgl. 3.3), dass die Populationsgröße ebenfalls einen großen Einfluss auf den Simulationsverlauf hat. Um die Ergebnisse (vgl. 4.2) unserer Simulationen vergleichbar zu halten, wurde auf eine Variierung der Populationsgröße verzichtet. Der Einfluss dieser Variablen sollte jedoch ebenfalls in weiteren Simulationen betrachtet werden. Zudem sollte auch der Einfluss der Lernversuchs-Anzahl genauer untersucht werden.

6. Quellen

Hinton, G.E. & Nowlan, S.J. (1987). How learning can guide evolution. Complex Systems, 1, 495–502.

Watson, R.A. (2007). Compositional evolution. MIT Press

Watson, R.A., Pollack, J. (2001). Symbiotic Composition and Evolvability. - Scientific Figure. Verfügbar über: https://www.researchgate.net/figure/A-section-through-a-scale-invariant-HIFF-fitness-landscape_fig1_221531339 [zugegriffen am 9 Jul, 2022]

Extended Evolutionary Synthesis. How the EES differs from the Modern Synthesis. Verfügbar über: https://extendedevolutionarysynthesis.com/about-the-ees/#how-the-ees-differs-from-the-modern-synthesis [zugegriffen am 09.07.2022]