Variationsoperatoren

Jens Kosiol

(Teilweise angelehnt an den Foliensatz von Eiben und Smith)

Überblick

- Wir entwickeln Variationsoperatoren (Mutation und Rekombination) für die Repräsentationen aus dem letzten Kapitel:
  - Strings über endlichem Alphabet (Verallgemeinerung von Bitstrings)
  - Permutationen
  - Bäume und Graphen
  - Vektoren natürlicher oder reeller Zahlen
- Wir betrachten Eigenschaften der eingeführten Operatoren:
  - Korrektheit: Bewahrt die Anwendung des Operators die gewählte Repräsentation?
  - Vollständigkeit: Bleiben alle Teile des Suchraums erreichbar?
  - Wie verteilen sich die berechenbaren Nachkommen?

Mutationsoperatoren für Strings

Sei Σ ein (endliches) Alphabet und Σ^n (Strings der Länge n über Σ) der gewählte Suchraum.

Typischer Mutationsoperator:

- Ändere jedes Gen unabhängig mit Mutationswahrscheinlichkeit p_m (p_m wird meistens zwischen 1/Populationsgröße und 1/Länge der Kodierung gewählt)
- Wird das i-te Gen mit Wert $\sigma_i \in \Sigma$ zur Mutation gewählt, so wird es unabhängig gleichverteilt auf einen Wert $\Sigma \setminus \{\sigma_i\}$ gesetzt

Beispiel Stringmutation

Sei $\Sigma = \{0,1,2\}$ das betrachtete Alphabet und kodiere jede Zahl einen möglichen Zug des Spiels “Schere, Stein, Papier”. Strings der Länge 9 kodieren dann eine Spielstrategie mit einem Gedächtnis der Länge 1.

Ist ein Gen zur Mutation ausgewählt, wird es mit jeweils 50% Wahrscheinlichkeit auf einen der verbleibenden Werte gesetzt.

https://bigbangtheory.fandom.com/de/wiki/Stein,_Papier,_Schere,_Echse,_Spock

Beispiel:

Elternchromosom: $[0,2,1,2,2,1,0,1,1]$

Kind: $[1,1,1,2,2,1,2,1,1]$

- Bitweise wird mit Wahrscheinlichkeit p_{m} eine Mutation durchgeführt. Hier Positionen 1,2 und 7 zur Mutation gewählt.
- Das Allel 0 an Position 1 wird mit je einer Wahrscheinlichkeit von 50% auf 1 bzw. 2 gesetzt; hier dann auf 1. Analog für Positionen 2 und 7.

Eigenschaften Stringmutation I

Sei Σ ein (endliches) Alphabet, Σ^n der betrachtete Suchraum und p_m die gewählte Mutationswahrscheinlichkeit. Dann hat der eben eingeführte Mutationsoperator auf Elementen von Σ^n die folgenden Eigenschaften:

- Korrektheit: Anwendung des Mutationsoperators auf Elemente aus Σ^n erzeugt wieder Elemente in Σ^n.
- Vollständigkeit: Jedes Chromosom kann durch die Anwendung von Mutationen in jedes andere überführt werden, mit geringer Wahrscheinlichkeit sogar in einem Schritt. Sind σ und σ' zwei Chromosomen mit Hammingdistanz d, so überführt eine Mutation σ zu σ' mit Wahrscheinlichkeit

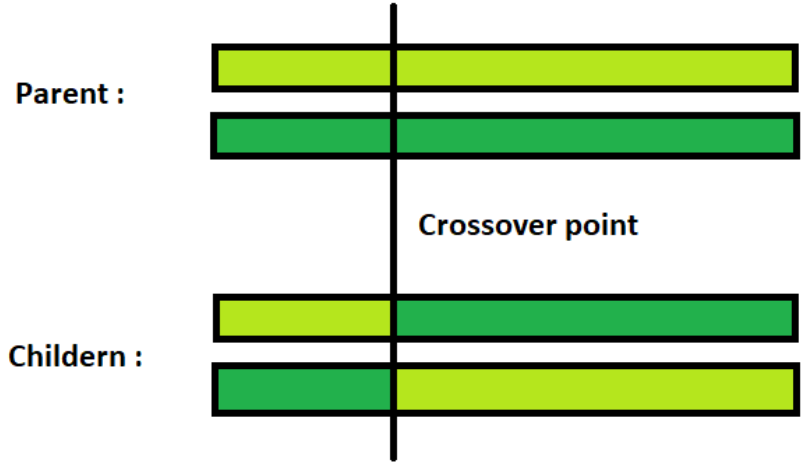
$$\left(\frac{p_\mathrm{m}}{|\Sigma|-1}\right)^d \times (1-p_\mathrm{m})^{n-d}$$

Eigenschaften Stringmutation II

Sei Σ ein (endliches) Alphabet, Σ^n der betrachtete Suchraum und p_m die gewählte Mutationswahrscheinlichkeit. Dann genügt der eben eingeführte Mutationsoperator auf Elementen von Σ^n den folgenden Verteilungen:

- Zu entscheiden, ob ein einzelnes Gen mutiert wird oder nicht, ist ein Bernoulli-Experiment mit Erfolgswahrscheinlichkeit p_m, der gesamte Vorgang ein Bernoulli-Prozess.
- Die Anzahl der geänderten Gene genügt also der Binomialverteilung.
  - Die Wahrscheinlichkeit, dass die Anwendung des Mutationsoperators genau k Gene ändert, beträgt daher $\binom{n}{k} p_\mathrm{m}^k (1 - p_\mathrm{m})^{n-k}$.
  - Die erwartete Anzahl der geänderten Gene ist np_m.
  - Die Varianz beträgt $np_\mathrm{m}(1 - p_\mathrm{m})$.

Rekombinationsoperatoren für Strings: 1-Punkt Crossover



https://www.geeksforgeeks.org/crossover-in-genetic-algorithm/

- Für Strings der Länge m bestimme eine zufällige natürliche Zahl $1 \leq p \leq m - 1$.
- Zerschneide die beiden Elternstrings jeweils hinter der Position p.
- Die Kinder entstehen durch kreuzweises Rekombinieren der Eltern.

Großer Nachteil positional bias:

- Gene, die nahe beieinander liegen, haben eine hohe Wahrscheinlichkeit, gemeinsam vererbt zu werden.
- Gene von den gegenüberliegenden Enden eines Strings können nie zusammen an ein Kind vererbt werden.

Rekombinationsoperatoren für Strings: n-Punkt Crossover

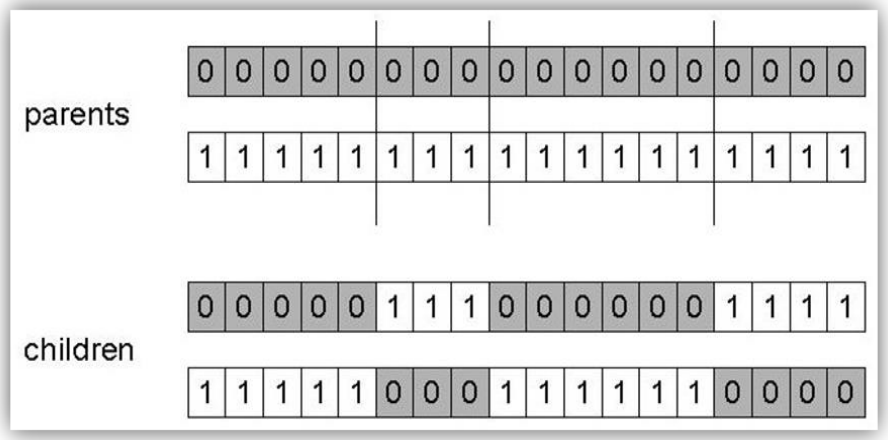


Illustration eines 3-Punkt Crossovers

Der n-Punkt Crossover verallgemeinert den 1-Punkt Crossover:

- n verschiedene zufällige Zahlen $p_1, \ldots, p_n$ zwischen 1 und m (Stringlänge) werden gewählt.
- Die Elternchromosomen werden an den Positionen $p_1, \ldots, p_n$ zerschnitten und die Kinder durch abwechselndes Kombinieren der Teilstrings der Eltern erzeugt.
- Der positional bias wird so reduziert, aber noch nicht vollständig vermieden.

Rekombinationsoperatoren für Strings: Uniform Crossover

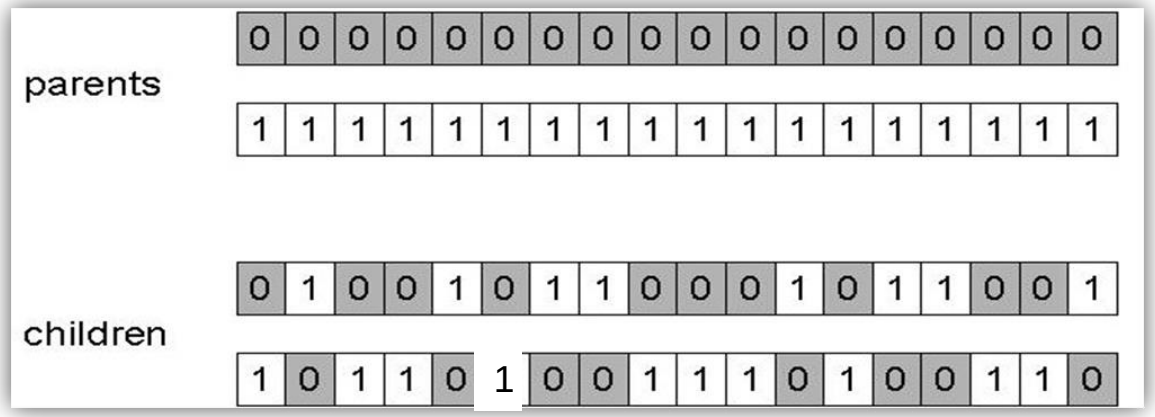


Illustration uniform crossover

Uniform crossover vermeidet den positional bias auf Kosten eines distributional bias:

- Es gibt eine starke Tendenz, auf jedes Kind ca. 50 % der Gene zu vererben (für $p_{\text{uni}} = 0.5$).
- Die Tendenz, koadaptierte Gene gemeinsam an ein Kind zu vererben, ist gering.

Uniform crossover verteilt die einzelnen Gene unabhängig voneinander auf die Kinder:

- Eine Wahrscheinlichkeit p_{uni} wird gewählt (meist 0.5).
- Das erste Elternchromosom wird positionsweise durchlaufen. Das erste Kind erbt jeweils mit Wahrscheinlichkeit p_{uni} das entsprechende Allel auf dieser Position und mit Wahrscheinlichkeit $1 - p_{\text{uni}}$ das Allel des zweiten Elternteils an dieser Position.
- Das zweite Kind erbt jeweils das Allel, das das erste nicht geerbt hat.

Eigenschaften Rekombination auf Strings

Die vorgestellten Crossover-Operatoren verändern nicht die Allelfrequenz einer Population – sie mischen lediglich die vorkommenden Allele neu unter den Chromosomen der Population:

Sei $\Sigma = \{\sigma_1, \ldots, \sigma_l\}$ ein endliches Alphabet, P eine Population über Σ^kund für $1 \leq i \leq l, 1 \leq j \leq k$ bezeichne $\mathrm{fr}_j(\sigma_i, P)$ die Frequenz von σ_i an der Position j in P. Wählt man zwei beliebige (nicht notwendig verschiedene) Chromosomen $p_1, p_2 \in P$, wendet auf diese Crossover an (n-Punkt oder uniform) und erhält Nachkommen o_1, o_2, so gilt $\mathrm{fr}_j(\sigma_i, P) = \mathrm{fr}_j(\sigma_i, P')$ für alle i, j, wobei $P' = P \setminus \{p_1, p_2\} \cup \{o_1, o_2\}$.

Probleme klassischer Variationsoperatoren auf Permutationen

Mutation

$(1,2,3,4,5,6)$

Mutation mit zufälligem Flip der Positionen 1 und 5.

$(6,2,3,4,2,6)$

Crossover

Eltern: $(1,2,3,4,5,6)$

$(6,5,4,3,2,1)$

1-Punkt Crossover mit Crossoverpunkt an Position 4.

Offspring: $(1,2,3,4,2,1)$

$(6,5,4,3,5,6)$

Anwenden der klassischen Variationsoperatoren für Strings führt zu Chromosomen, die keine Permutationen darstellen!

Variationsoperatoren auf Permutationen

Welche Information soll an die Nachkommen vererbt werden?

- Relative Ordnungsstruktur? (Wenn in einem Elternteil die 1 nach der 5 liegt, soll diese Information an einen Nachkommen vererbt werden können.)
- Position von Elementen? (Wenn in einem Elternteil die 1 an der 3. Position liegt, soll diese Information an einen Nachkommen vererbt werden können.)
- Kanten zwischen Elementen? (Wenn 1 und 5 in einem Elternteil benachbart sind, soll diese Information an einen Nachkommen vererbt werden können.)

Gerade im Zusammenhang mit dem TSP wurden viele Variationsoperatoren auf Permutationen entworfen, die unterschiedliche Arten von Informationen vererben.

Mutationsoperatoren für Permutationen

Für Permutationen wurden verschiedene Mutationsoperatoren vorgeschlagen:

- Swap Mutation: Vertausche die Allele zweier Gene miteinander (auch exchange mutation genannt).
- Insert Mutation: Ein Allel wird hinter einem anderen eingeordnet.
- Scramble Mutation: Die Positionen ausgewählter Allele werden gemischt.
- Simple Inversion Mutation: Zwei Positionen werden gewählt und die Reihenfolge der dazwischen liegenden Allele umgedreht.
- Displacement und Inversion Mutation: Ein Substring wird gewählt und (in umgekehrter Reihenfolge) an zufälligem Ort neu platziert.

Swap Mutation

Wähle zufällig gleichverteilt zwei Allele und vertausche ihre Positionen (oder Positionen und vertausche ihre Allele):

Die Allele 2 und 5 wurden zufällig ausgewählt und tauschen ihre Positionen.

Swap Mutation hat moderate Auswirkungen sowohl auf die Reihenfolge als auch auf die Nachbarschaftsstruktur (meistens 4 Verbindungen geändert).

Insert Mutation

Wähle zufällig gleichverteilt zwei Allele und positioniere das zweite direkt hinter dem ersten; die restlichen Allele werden entsprechend geshiftet.

Die Allele 2 und 5 wurden zufällig gewählt und das Allel 2 hinter dem Allel 5 positioniert.

Insert Mutation hat moderate Auswirkungen sowohl auf die Reihenfolge als auch auf die Nachbarschaftsstruktur (meistens 4 Verbindungen geändert).

Scramble Mutation

Wähle zufällig eine Teilmenge der Gene (inklusive zufälliger Wahl der Größe) und mische die Reihenfolge der Allele der entsprechenden Positionen zufällig neu.

Zufällige Wahl der Gene 2 bis 5 und anschließendes Mischen der Reihenfolge der entsprechenden Allele.

Scramble Mutation hat häufig gravierendere Auswirkungen sowohl auf die Reihenfolge als auch auf die Nachbarschaftsstruktur.

Simple Inversion Mutation

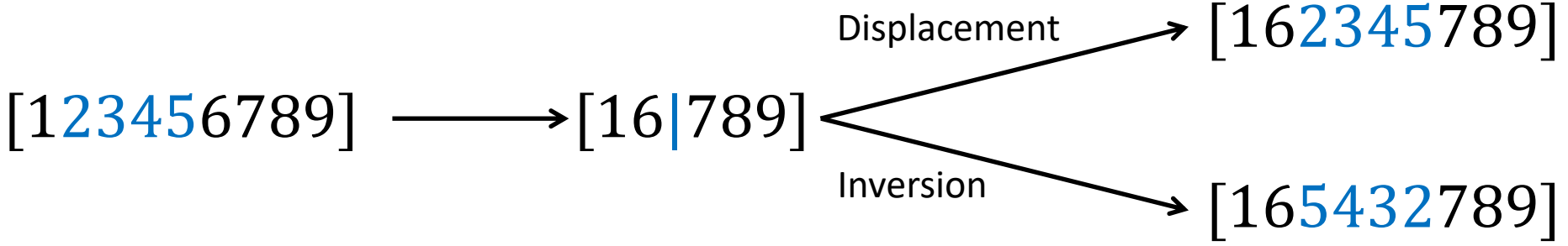
Wähle zufällig gleichverteilt zwei Allele und invertiere die Reihenfolge des davon aufgespannten Substrings.

Die Allele 2 und 5 wurden gewählt und der dazwischen liegende Substring invertiert.

Simple Inversion Mutation hat häufig größere Auswirkungen auf die Reihenfolge, aber nur moderate Auswirkungen auf die Nachbarschaftsstruktur (2 Verbindungen geändert).

Displacement und Inversion Mutation

Wähle zufällig gleichverteilt zwei Allele. Füge den davon aufgespannten Substring an zufälliger Position neu ein (für Inversion Mutation: in umgekehrter Reihenfolge).



Die Allele 2 und 5 wurden gewählt und der dazwischen liegende Substring neu platziert.

Displacement und Inversion Mutation bewahren große Teile der Nachbarschaftsstruktur, ändern aber vergleichsweise viele Positionen und relative Reihenfolge (gerade Inversion Mutation).

Mutationswahrscheinlichkeit für Permutationen

- Alle sechs betrachteten Mutationsoperatoren auf Permutationen arbeiten nicht positionsweise, sondern auf einem vollständigen Chromosom.
- Es kann also nicht mehr positionsweise entschieden werden, ob ein Gen mutiert wird oder nicht, sondern die Entscheidung muss einmal für das gesamte Chromosom getroffen werden.
- Die Mutationswahrscheinlichkeit p_{m} kodiert nun also, mit welcher Wahrscheinlichkeit ein Chromosom überhaupt mutiert werden soll (und wird folglich meist deutlich höher angesetzt).
- Wenn verschiedene Mutationsoperatoren in einem Algorithmus eingesetzt werden sollen, muss man zusätzlich ein Verfahren ergänzen, um den jeweils anzuwendenden Operator auszuwählen (z.B.: Wenn ein Chromosom zur Mutation ausgewählt wurde, wird anschließend zufällig gleichverteilt einer der verfügbaren Mutationsoperatoren gewählt und mit diesem die Mutation durchgeführt.).

Eigenschaften Mutation auf Permutationen

Sei M eine (endliche) Menge und der betrachtete Suchraum seien die Mutationen über M. Dann haben alle eingeführten Mutationsoperatoren auf Permutationen die folgenden Eigenschaften:

- Korrektheit: Anwendung eines der Mutationsoperatoren überführt eine Permutation wieder in eine Permutation.
- Vollständigkeit: Jedes Chromosom kann durch die wiederholte Anwendung von Mutationen in jedes andere überführt werden. Scramble Mutation ist jedoch die einzige Mutationsart, wo es für jede zwei Chromosomen eine Wahrscheinlichkeit > 0 (wenn auch extrem gering) gibt, dass dies in einem Schritt geschehen kann.

Crossoveroperatoren für Permutationen

Auch hier gibt es viele Vorschläge; häufig entwickelt anhand des Traveling Salesman Problem.

- Order Crossover
- Partially Mapped Crossover
- Cycle Crossover
- Edge Recombination

Manchmal wird nur ein (statt zwei) Nachkomme berechnet.

Order Crossover (OX1)

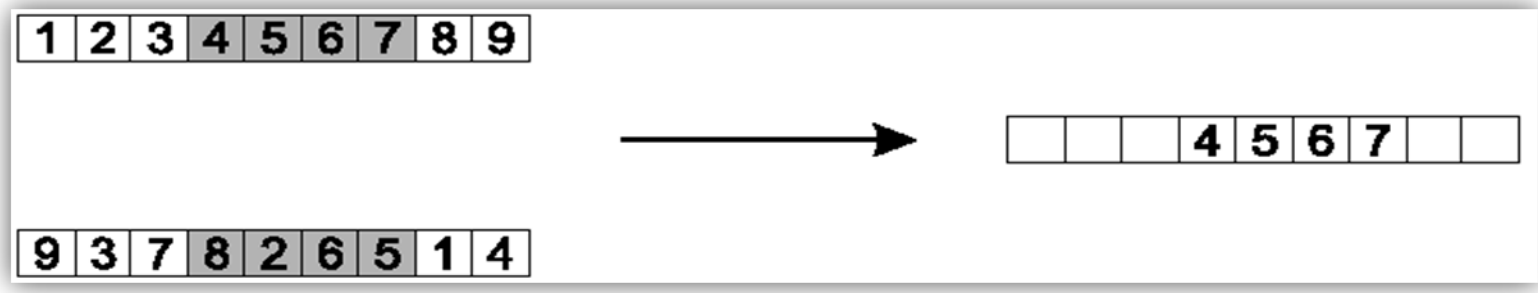
Ziel: Relative Ordnung von Elementen bewahren; entwickelt in [OSH87]

Vorgehen: Gegeben sind zwei Permutationen π_1 und π_2 der Länge $n = |M|$ einer Menge M.

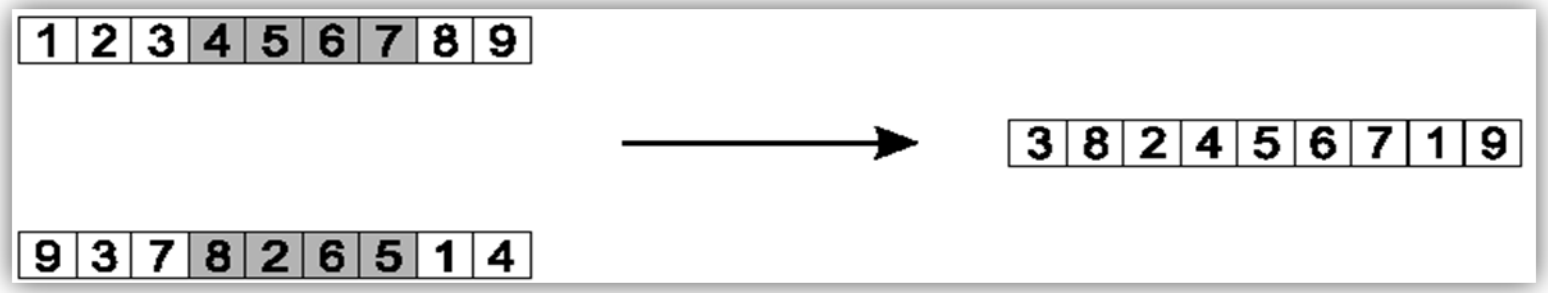
1. Bestimme zufällig gleichverteilt zwei Crossoverpunkte $1 \leq i < j \leq n$.
2. Kopiere die Allele an den Positionen i bis j aus π_1 an die gleichen Positionen im ersten Nachkommen o_1.
3. Fülle o_1, beginnend an Position $j + 1$, mit den Allelen aus π_2, auch beginnend an Position $j + 1$, auf. Hierbei
   1. werden Allele aus π_2, die bereits in o_1 vorkommen, übersprungen;
   2. wird jeweils bei Erreichen von Position n bei Position 1 fortgesetzt.
4. Der zweite Nachkomme wird analog mit vertauschten Rollen von π_1 und π_2 berechnet.

OX1 Beispiel

Schritte 1 und 2: Kopiere zufälligen Abschnitt aus erstem Elternteil



Schritt 3: Fülle mit Elementen aus zweitem Elternteil in der entsprechenden Reihenfolge auf



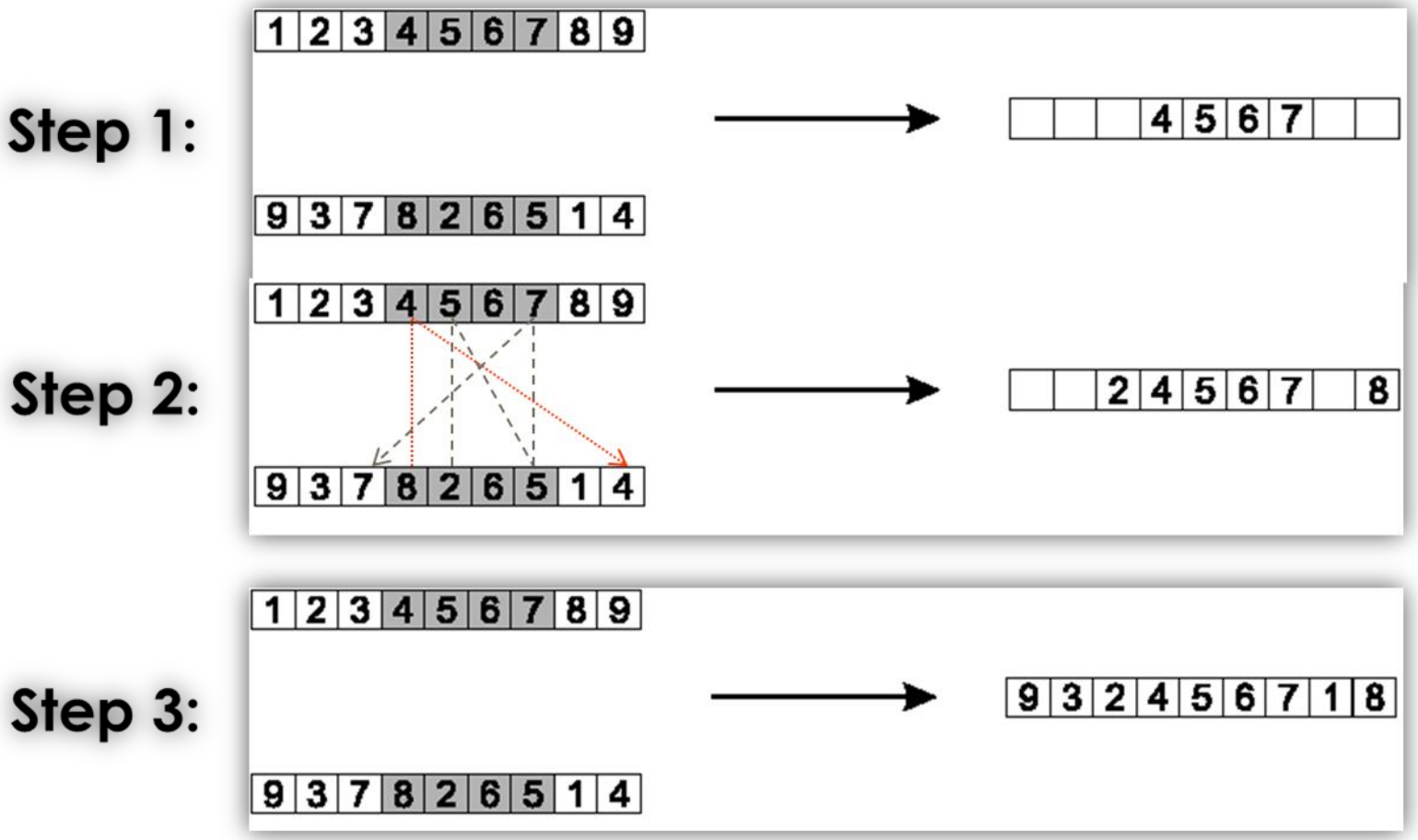
Partially Mapped Crossover (PMX)

Ziel: Position von Elementen bewahren; entwickelt in [GL85]

Vorgehen: Gegeben sind zwei Permutationen π_1 und π_2 der Länge $n = |M|$ einer Menge M.

1. Bestimme zufällig gleichverteilt zwei Crossoverpunkte $1 \leq i < j \leq n$. Kopiere die Allele an den Positionen i bis j aus π_1 an die gleichen Positionen im ersten Nachkommen o_1 und die aus π_2 an die gleichen Positionen in o_2. Bilde eine Bijektion b zwischen den Allelen aus π_1 und π_2 auf den Positionen i bis j.
2. Fülle die freien Positionen in o_1 mit dem Allel aus π_2 an dieser Position auf. Kommt das Allel a von einer Position k bereits in o_1 vor, so wird stattdessen $b(a)$ an dieser Stelle eingetragen. Kommt $b(a)$ bereits in o_1 vor, setze $a' := b(a)$ und fülle mit $b(a')$ auf (iterativ).
3. o_2 wird analog aufgefüllt.

PMX Beispiel



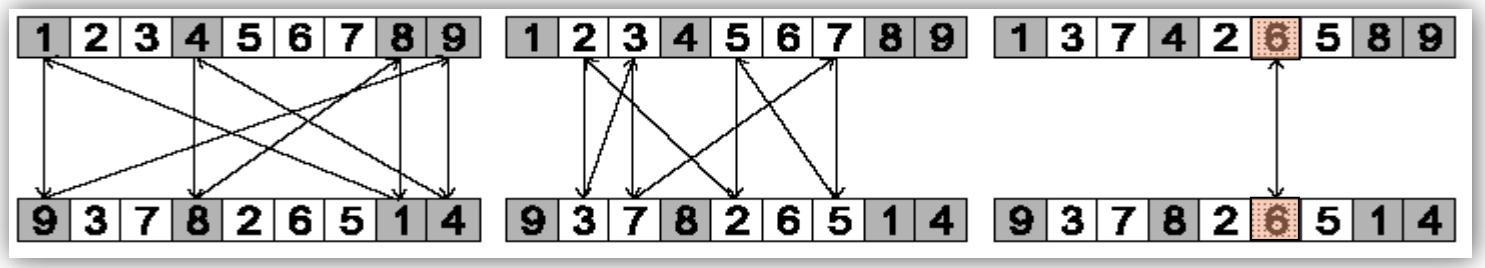
Cycle Crossover (CX)

Ziel: In den Nachkommen soll an jeder Position ein Allel stehen, das in einem der Elternteile an der entsprechenden Position steht; entwickelt in [OSH87].

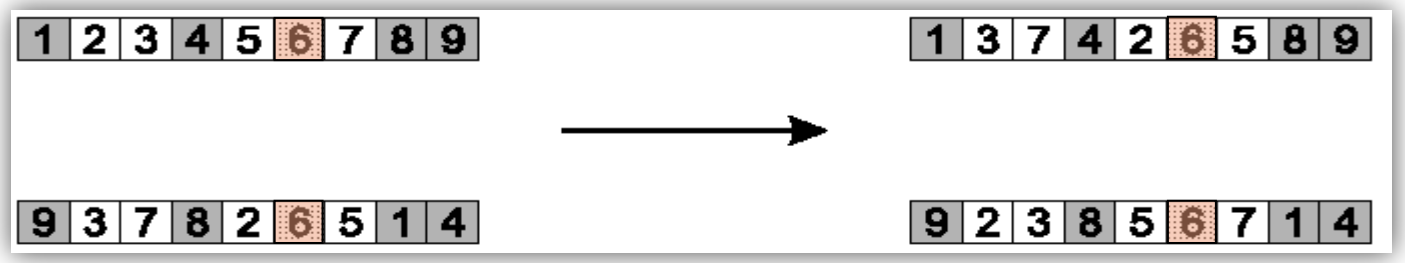
Vorgehen: Gegeben sind zwei Permutationen π_1 und π_2 der Länge $n = |M|$ einer Menge M.

- Füge abwechselnd, jeweils beginnend bei der ersten freien Position des Nachkommen, einen Zykel von Allelen aus π_1 bzw. π_2 ein.
- Berechnung eines Zykels aus π_1 (analog für π_2):
  1. Starte mit dem Allel aus π_1, das an der ersten freien Position des Nachkommen steht.
  2. Überprüfe, welches Allel a in π_2 an dieser Position steht.
  3. Finde die Position von Allel a in π_1 und füge diese Position dem Zykel hinzu.
  4. Wiederhole Schritte 2. und 3. bis der Zykelstart in π_1 wieder erreicht ist. Kopiere den entstehenden Zykel in den Nachkommen.
- Varianten:
  - Berechnung von zwei Nachkommen, indem Reihenfolge von π_1 und π_2 getauscht werden.
  - Berechnung von nur einem Zyklus aus π_1 (beginnend an Position 1) und Auffüllen der restlichen Positionen aus π_2.

CX Beispiel



1. Bestimmen von Zykeln



2. Kopieren von Zykeln in die Nachkommen

Edge Recombination (ER)

Ziele:

- Möglichst keine zufälligen Kanten erzeugen, also nur Kanten im Nachkommen erstellen, die in einem der Elternteile auftauchen. Kommen im Nachkommen etwa 2 und 5 nebeneinander zu stehen, sollten 2 und 5 möglichst auch in einem Elternteil benachbart sein.
- Möglichst gemeinsame Kanten an Nachkommen vererben. Wenn in beiden Eltern also 2 und 5 nebeneinander stehen, sollten sie dies auch im Nachkommen tun.

Hilfsmittel: Kantentabelle, die Überblick über die Verbindungen gibt.

Ursprung: Existiert in mehreren Varianten; hier nach [Whitley00].

ER Kantentabelle

Element	Edges	Element	Edges
1	2,5,4,9	6	2,5+,7
2	1,3,6,8	7	3,6,8+
3	2,4,7,9	8	2,7+, 9
4	1,3,5,9	9	1,3,4,8
5	1,4,6+		

Beispiel für Kantentabelle für Elternpaar [123456789] und [937826514];
doppelte Vorkommen sind per + markiert

ER Durchführung

Vorgehen: Gegeben sind zwei Permutationen π_1 und π_2 der Länge $n = |M|$ einer Menge M.

1. Erstelle Kantentabelle
2. Wähle zufällig gleichverteilt ein Startelement a aus M. a wird das Allel an Position 1 im Nachkommen; setze außerdem eine Variable `current element` auf a.
3. Lösche alle Vorkommen von `current element` aus den Kantenlisten Kantentabelle.
4. Inspiziere die Kantenliste von `current element`:
    1. Gibt es eine gemeinsame Kante, wird diese zum `current element` und wird außerdem an der nächsten freien Position im Nachkommen eingetragen.
    2. Sonst wird das Element gewählt, das die kürzeste Kantenliste hat.
    3. Gleichstände werden gleichverteilt zufällig aufgelöst.
5. Wiederhole 3. und 4. bis der Nachkomme berechnet ist. Falls zwischendrin eine leere Liste erreicht wird: Wiedereinsteig bei 2. mit einem zufälligen, noch nicht benutzten Element.

Edge Recombination Beispiel

Element	Edges	Element	Edges
1	2,5,4,9	6	2,5+,7
2	1,3,6,8	7	3,6,8+
3	2,4,7,9	8	2,7+, 9
4	1,3,5,9	9	1,3,4,8
5	1,4,6+		

Elternchromosomen:
[123456789] und [937826514]

Choices	Element selected	Reason	Partial result
All	1	Random	[*1*]
2,5,4,9	5	Shortest list	[*1 5*]
4,6	6	Common edge	[*1 5 6*]
2,7	2	Random choice (both have two items in list)	[*1 5 6 2*]
3,8	8	Shortest list	[*1 5 6 2 8*]
7,9	7	Common edge	[*1 5 6 2 8 7*]
3	3	Only item in list	[*1 5 6 2 8 7 3*]
4,9	9	Random choice	[*1 5 6 2 8 7 3 9*]
4	4	Last element	[*1 5 6 2 8 7 3 9 4*]

Formale Eigenschaften Crossover auf Permutationen

Gegeben sind zwei Permutationen π_1 und π_2 der Länge $n = |M|$ einer Menge M. Sei k die Länge des Strings zwischen den Crossoverpunkten und bezeichne o_1 und o_2 die Nachkommen.

- Wird ein vollständiger Teilstring in Nachkommen kopiert, wird an den entsprechenden Teil Information über absolute Position, Ordnung und Nachbarschaft vererbt.
- In OX1 bleibt in Nachkomme o_1 viel der relativen Ordnung der Elemente aus π_2 erhalten (und in o_2 die aus π_1). Ist der Startpunkt einer Permutation irrelevant (wie bei TSP), behalten mindestens $n - k$ Elemente ihre relative Ordnung.
- In PMX bleiben in o_1 viele der absoluten Positionen aus π_2 erhalten (und in o_2 die aus π_1); ca. $n - k$ Elemente behalten ihre absolute Position.
- In CX wird Information über die absolute Position vererbt: Im Schnitt haben etwa die Hälfte der Allele ihre Position aus π_1, die andere Hälfte ihre Position aus π_2.
- ER vererbt Informationen über die Kanten aus den Elternteilen an die Nachkommen.

Empirische Evaluation

Es gibt umfangreiche empirische Vergleiche der Operatoren, sehr häufig für das Traveling Salesman Problem und häufig mit Konzentration auf Crossover.

- Überblicksstudien z.B. in [SMcDMWW91, Potvin96, LKMID99]
- Crossover:
  - Für TSP gilt ER > OX > PMX > CX (für die hier besprochenen Operatoren)
  - OX ist nicht viel schlechter als ER; der Abstand zu den anderen Operatoren ist deutlicher
  - Auf anderen Problemen kann auch CX am besten abschneiden [z.B. beim genetischen Lernen Bayesscher Netzwerke; LKMID99]
- Mutationen: Von den hier besprochenen Mutationsoperatoren waren für TSP Displacement, Inversion und Insertion Mutation besonders gut geeignet [LKMID99].

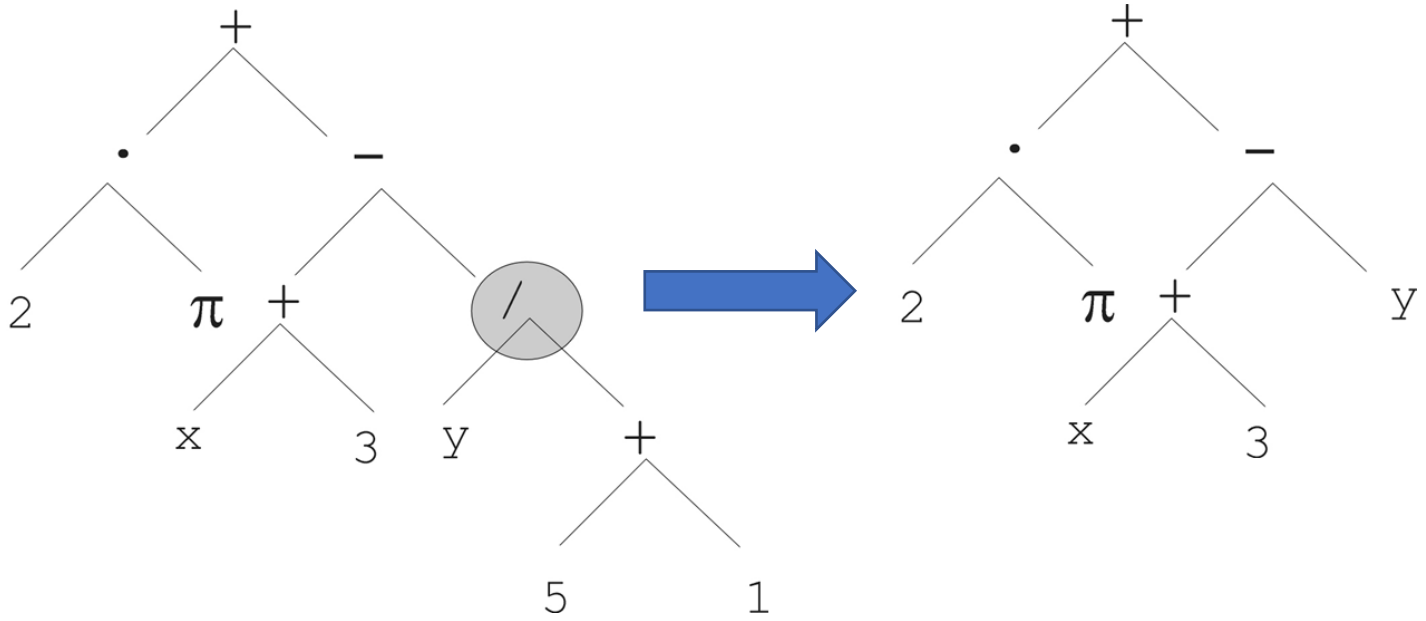
Variationsoperatoren auf Graphen und Bäumen

Der Entwurf von Variationsoperatoren für Graphen und Bäume stellt vor neue Herausforderungen:

- Es gibt keine („natürliche“) Entsprechung für Begriffe wie „Gen“, „Position“ oder „Allel“ mehr.
- Für viele Optimierungsprobleme, die wir als Graphen/Bäume kodieren wollen, ist ein nicht-endlicher Suchraum eine gute Wahl.
  ⇒ Wir benötigen Variationsoperatoren, die die Größe von Chromosomen ändern können.
- Es gibt Strukturen, die durch die Operatoren bewahrt werden sollten:
  - Bäume sollten nicht zu Graphen entarten.
  - Label von Knoten und Kanten transportieren oft eine Semantik, die Bedingungen an die Struktur nach sich zieht (z.B.: Stelligkeit eines Funktionssymbols bestimmt Anzahl der ausgehenden Kanten).

Mutation auf Bäumen

Häufigster Mutationsoperator auf Bäumen: Tausche einen zufällig ausgewählten Unterbaum durch einen zufällig generierten Baum aus.



$$2\pi + \left((x+3) - \frac{y}{5+1}\right) \rightarrow$$
$$2\pi + ((x+3) - y)$$

Der Unterbaum mit / als Wurzel wird durch den zufällig generierten Unterbaum y ersetzt.

Mutation auf Bäumen II

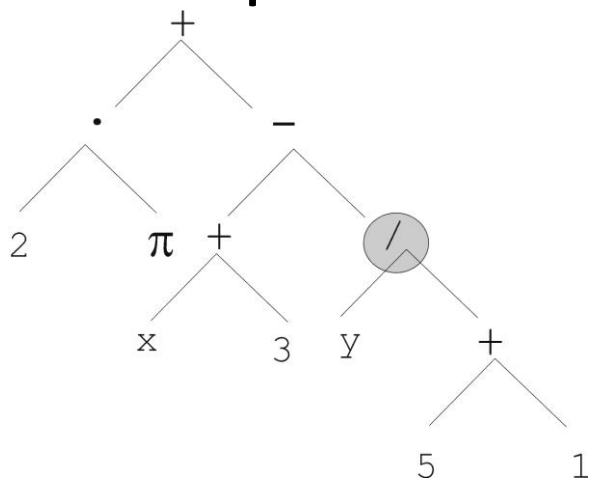
- Mutation auf Bäumen hat mehrere Komponenten
  - Mutationswahrscheinlichkeit p_{m}, die bestimmt, mit welcher Wahrscheinlichkeit überhaupt eine Mutation auf dem Baum stattfindet.
  - Verfahren, die Wurzel des zu ersetzenden Unterbaums zufällig zu bestimmen.
  - Verfahren, einen zufälligen Baum (mit den passenden Labels) zu generieren.
- In praktischen Anwendungen von evolutionären Algorithmen auf Bäumen wurde oft ohne Mutation ($p_{\mathrm{m}} = 0$) oder mit extrem geringer Mutationswahrscheinlichkeit ($p_{\mathrm{m}} = 0.05$) gearbeitet (Crossover als entscheidender Variationsoperator).

Crossover auf Bäumen

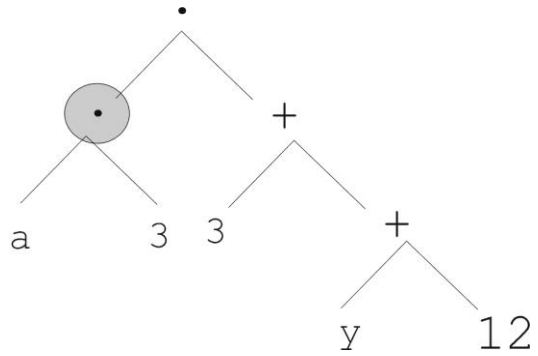
Häufigster Crossoveroperator auf Bäumen: Austausch von Unterbäumen zwischen zwei Bäumen.

- Crossover auf Bäumen hat mehrere Komponenten:
  - Crossoverwahrscheinlichkeit p_c, die festlegt, mit welcher Wahrscheinlichkeit Crossover auf einen Baum angewendet wird.
  - Verfahren, um die Wurzeln der auszutauschenden Unterbäume zufällig zu bestimmen.

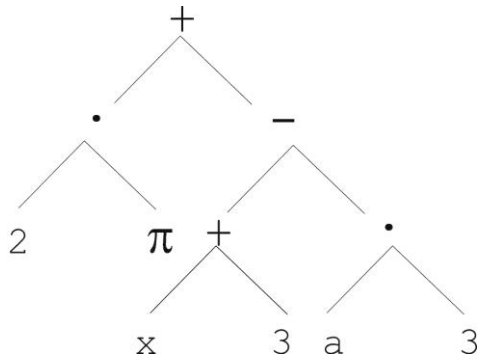
Beispiel Crossover auf Bäumen



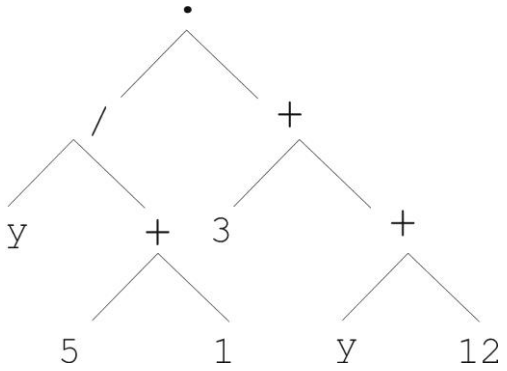
Parent 1



Parent 2



Child 1



Child 2

Formale Eigenschaften von Variationsoperatoren auf Bäumen

- Beide Variationsoperatoren können die Größe von Chromosomen ändern.
- Beide Variationsoperatoren sind korrekt in dem Sinne, dass sie Bäume wieder in Bäume überführen.
- Beide Variationsoperatoren bewahren die Anzahl an ausgehenden Kanten eines Knoten, bewahren also die Stelligkeit von Operatoren in Fällen, wo die Knoten Operatoren repräsentieren.

Grundprinzipien von Mutationen von Graphen

Grundidee hinter der Mutation von Graphen ist die Anwendung von Graphprogrammen, die (geringfügige) Modifikationen an Graphen durchführen.

- Ein Programm spezifiziert die Änderung von (wenigen) Elementen: Hinzufügen und/oder Löschen von Knoten und Kanten; Ändern von Attributwerten; Ändern von Labeln
- Geeignete Graphprogramme müssen für jedes Optimierungsproblem neu entworfen werden.
  - Häufig sind mehrere Programme nötig, um den Suchraum abdecken zu können.
  - Ein Graphprogramm ist (meist) spezifisch für ein Optimierungsproblem, dort aber für jede konkrete Probleminstanz einsetzbar.
  - Gegeben ein Graph kann ein passendes Graphprogramm (meist) an verschiedenen Orten in diesem Graphen angewendet werden.

Graphtransformationsregeln

Ein Ansatz zur Entwicklung von Graphprogrammen ist die regelbasierte Modifikation von Graphen („graph rewriting“).

Es gibt unterschiedliche Semantiken, aber grundsätzlich gilt: Eine Graphersetzungsregel

- spezifiziert einen Kontext, in dem sie angewendet werden kann;
- deklariert Änderungsaktionen (Hinzufügen und Löschen von Elementen, Ändern von Attributwerten und ggf. Labels);
- kann auf einen gegebenen Graphen im Normalfall an unterschiedlichen „Orten“ angewendet werden (Ansätze: Untergraphen, die dem spezifizierten Kontext entsprechen).

Durch Verkettung der Ausführung von Graphtransformationsregeln können (kleine) Graphprogramme entstehen, die einen Graphen modifizieren.

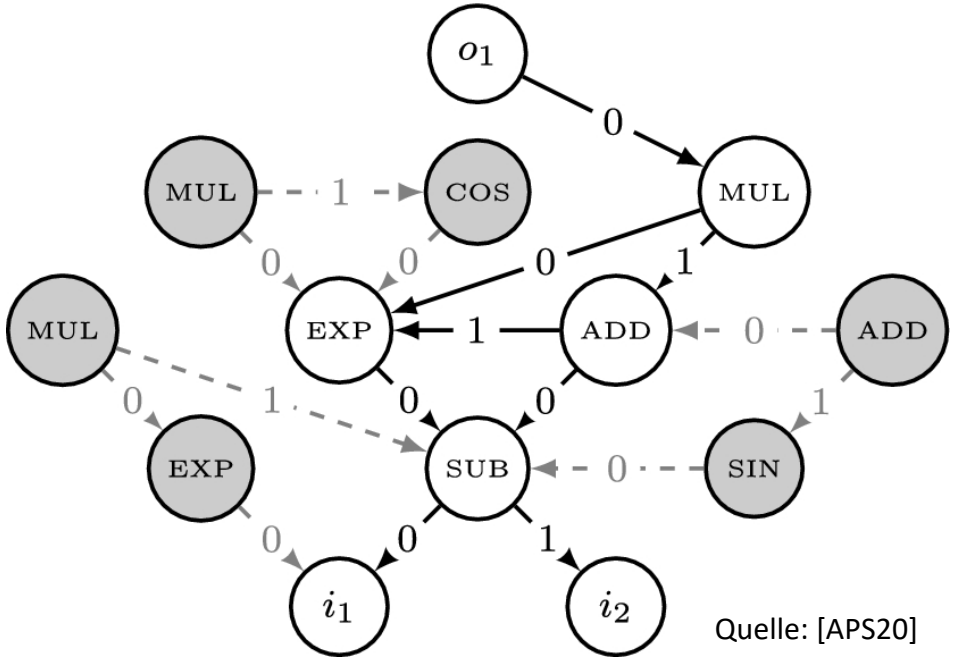
Ablauf Mutation von Graphen

1. Es muss eine (domänenspezifische) Auswahl an Graphprogrammen oder Graphtransformationsregeln getroffen werden (Wiederverwendung, Generierung oder händischer Entwurf).
2. Die Mutationswahrscheinlichkeit p_m bestimmt wieder, mit welcher Wahrscheinlichkeit ein Graph überhaupt mutiert wird.
3. Ist ein Graph zur Mutation ausgewählt, muss entschieden werden, welches Programm bzw. welche Regel(sequenz) an welchem Ansatz im Graphen angewendet wird. Zum Beispiel:
   - Unter *allen* Ansätzen *aller* verfügbaren Programme/Regeln wird zufällig gleichverteilt ausgewählt.
   - Unter den verfügbaren Programmen/Regeln wird zufällig gleichverteilt ausgewählt; für diese Regel wird unter den zur Verfügung stehenden Ansätzen zufällig gleichverteilt ausgewählt (iterativ, bis eine Regel angewendet wurde).
   - Mögliche Alternative zur Verwendung der Mutationswahrscheinlichkeit p_m: Zunächst (z.B. zufällig Bernoulli- oder Poisson-verteilt) bestimmen, wie viele Mutationen durchgeführt werden sollen.

Graphmutation: Beispiel 1

Knotenmutation für Symbolic Regression in [APS20]:

1. Wähle zufällig gleichverteilt einen Knoten, der eine Funktion repräsentiert, und ändere die repräsentierte Funktion auf ein anderes Symbol (z.B.: ADD → EXP).
2. Falls nötig, repariere die Stelligkeit des geänderten Knoten:
   - Bei zu vielen ausgehenden Kanten, lösche zufällig gleichverteilt Kanten, bis erlaubte Anzahl erreicht.
   - Bei zu wenig ausgehenden Kanten, füge zufällig gleichverteilt Kanten zu erlaubten Knoten ein, bis benötigte Anzahl erreicht; erlaubte Knoten: Knoten, zu denen Kanten zeigen können, ohne dass ein Zykel entsteht.
3. Ordne die Reihenfolge der ausgehenden Kanten des geänderten Knoten zufällig neu.

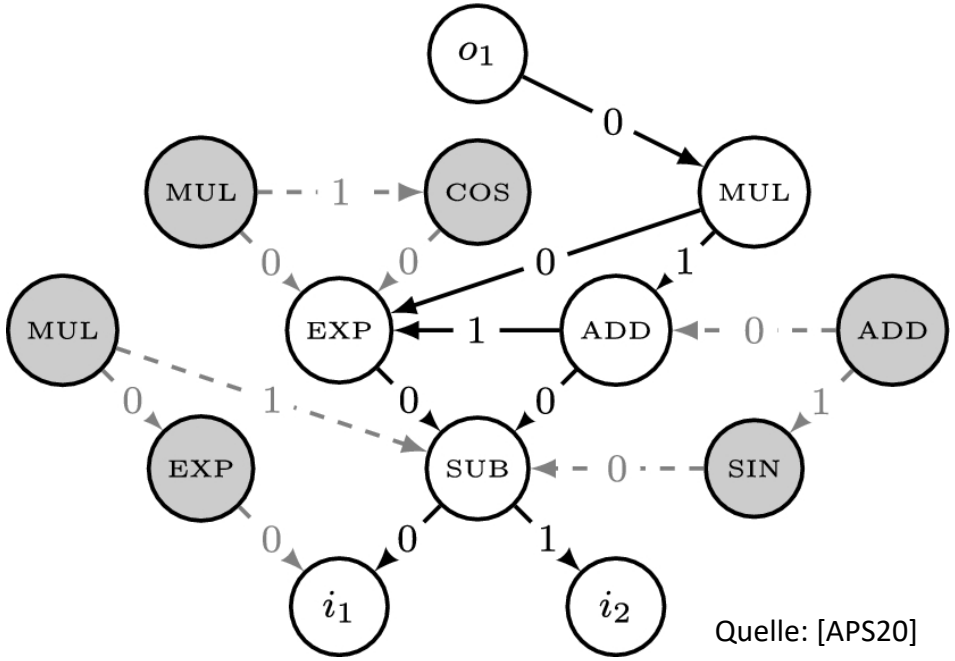


Quelle: [APS20]

Graphmutation: Beispiel 1 (Fortsetzung)

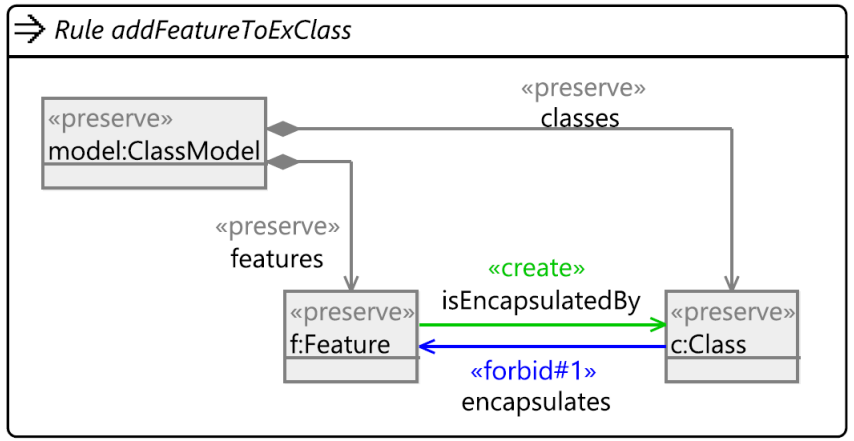
Kantenmutation für Symbolic Regression in [APS20]:

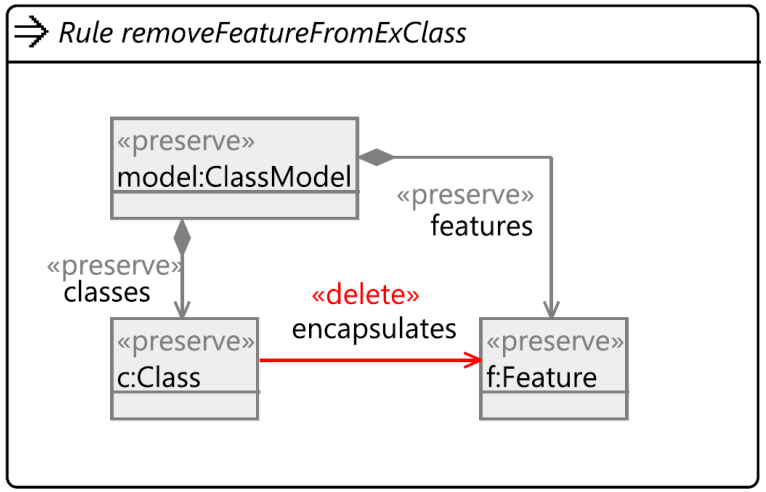
1. Wähle zufällig gleichverteilt eine Kante, die geändert werden soll.
2. Lenke die Kante auf einen beliebigen erlaubten Knoten um; unter den erlaubten Knoten wird zufällig gleichverteilt ausgewählt (erlaubte Knoten: Knoten, zu denen Kante zeigen kann, ohne dass ein Zykel entsteht).



Quelle: [APS20]

Notation für Graphersetzungsregeln



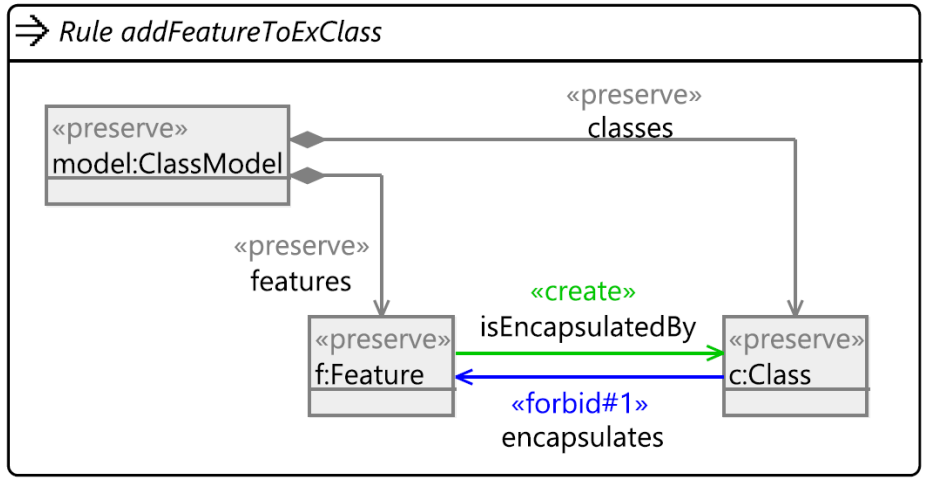


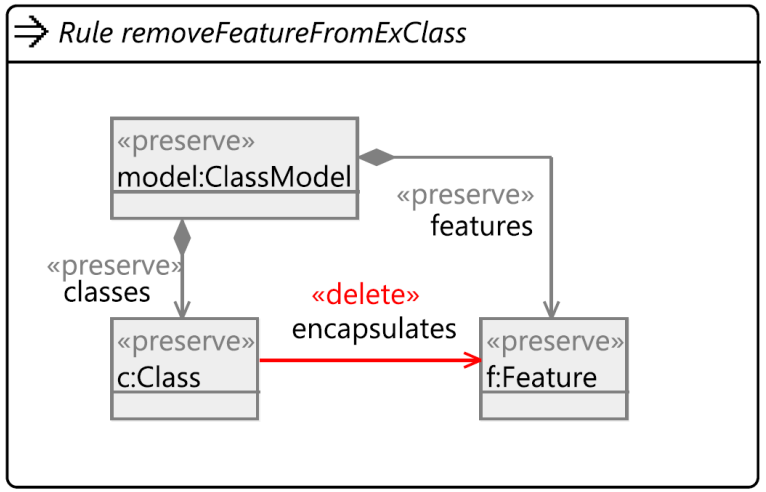
Kompakte, graphische Notation für Graphersetzungsregeln:

- Regel wird als ein Graph präsentiert, in dem Annotationen und Farben die Rollen der Elemente klären.
- Anwendung auf einen Graphen G:
  1. Es muss ein Untergraph in G gefunden werden, der den zu erhaltenden und zu löschenden Elementen aus der Regel (die grauen und roten Elemente) entspricht; die verbotenen Elemente (blau) dürfen dort nicht existieren.
  2. Die dadurch als zu löschend festgelegten Elemente werden aus G gelöscht.
  3. Die zu erzeugenden Elemente (grün) werden an der gewählten Stelle hinzugefügt.

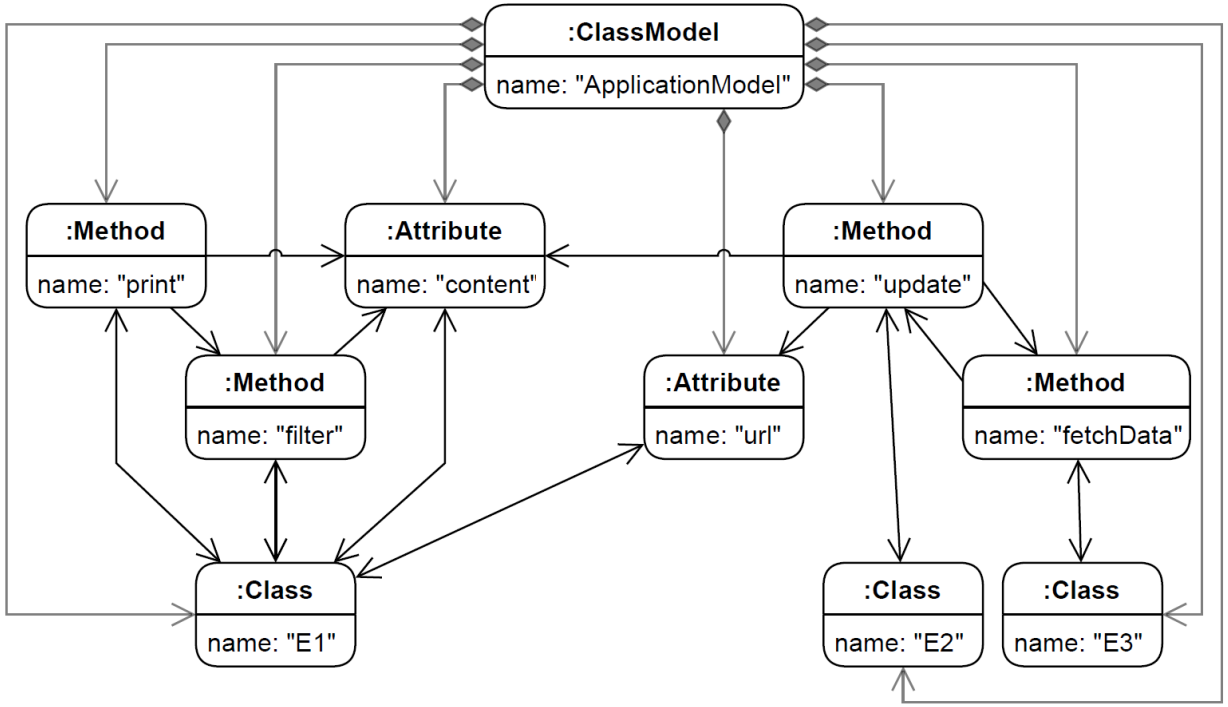
Quelle: [JKLT23]

Beispiel Anwendung Graphersetzungsregeln





Quelle: [JKLT23]



Quelle: [KJT23]

Eigenschaften von Graphmutationen

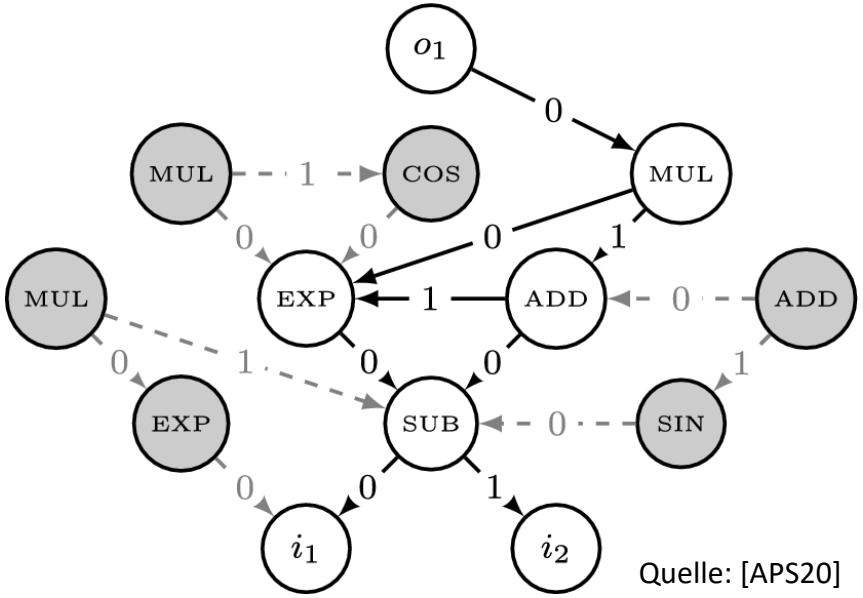
Gegeben ein Optimierungsproblem mit einer Graphkodierung der Lösungen, dann ist eine Menge von Graphmutationsoperatoren (Programme oder Regeln)

- vollständig, falls von jedem Graph G aus dem Suchraum aus jeder Graph H, der die Nebenbedingungen des Optimierungsproblems erfüllt, durch (eine Sequenz von) Mutationen erreicht werden kann;
- korrekt, falls jeder Graph G aus dem Suchraum, der die Nebenbedingungen des Optimierungsproblems erfüllt, durch die Anwendung von Mutationen in einen Graphen H überführt wird, der auch die Nebenbedingungen erfüllt (es können keine Verletzungen von Nebenbedingungen eingeführt werden).

Korrektheit und Vollständigkeit in [APS20]

Vollständigkeit:

- Die Knoten- und Kantenmutation aus [APS20] ist *nicht* vollständig: Die Knotenzahl bleibt stabil.
- Es gibt aber eine eingeschränkte Vollständigkeit: Bis zur fixen Größe des Graphen sind alle Phänotypen (Funktionen) erreichbar.

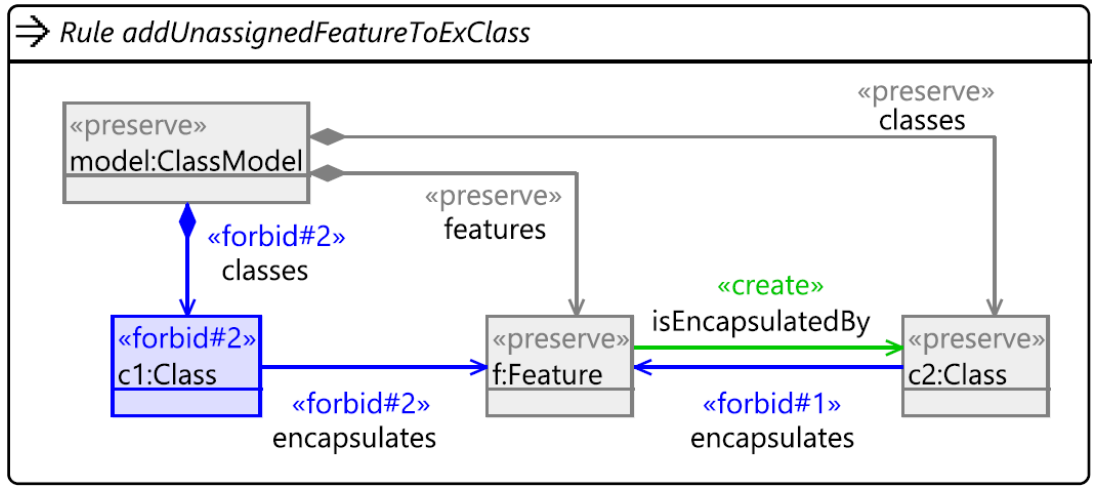


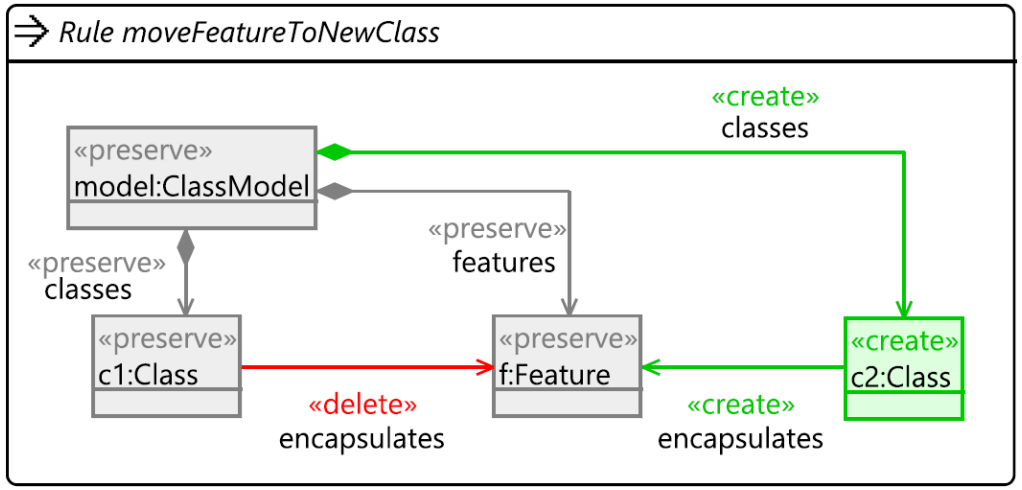
Quelle: [APS20]

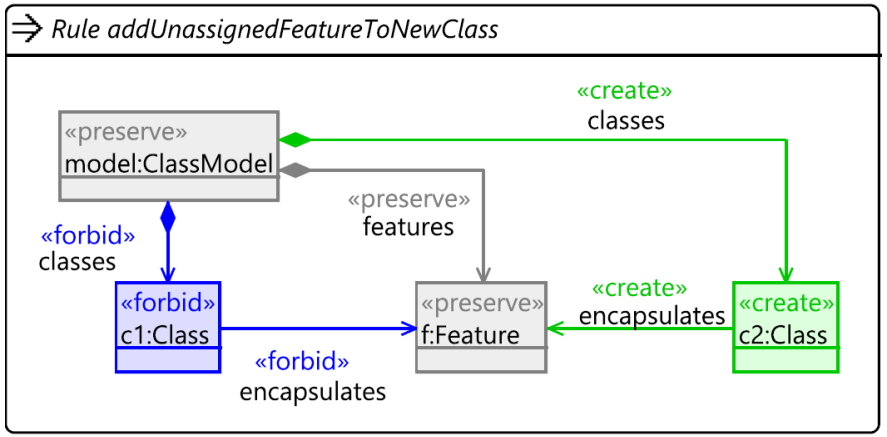
Korrektheit:

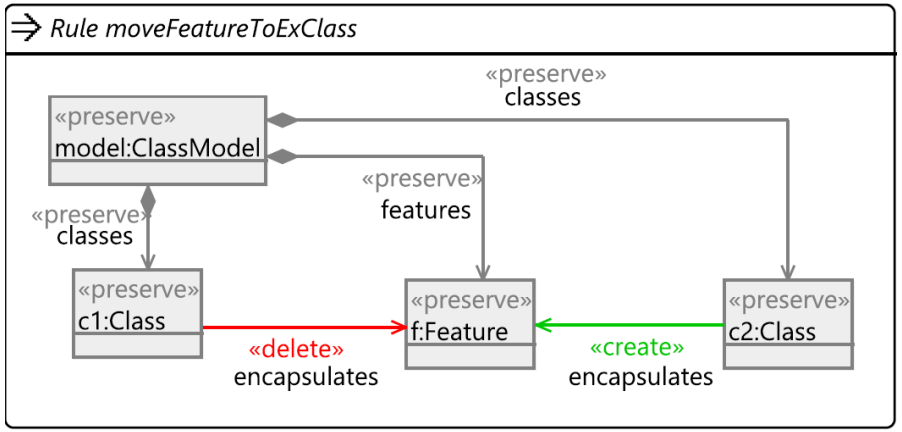
- Nebenbedingungen waren vollständiges Labeling, Zykelfreiheit und korrekte Anzahl ausgehender Kanten (siehe Folie 132).
- Knoten- und Kantenmutation aus [APS20] ist korrekt: Die Programme sind darauf ausgelegt, diese Bedingungen zu bewahren.

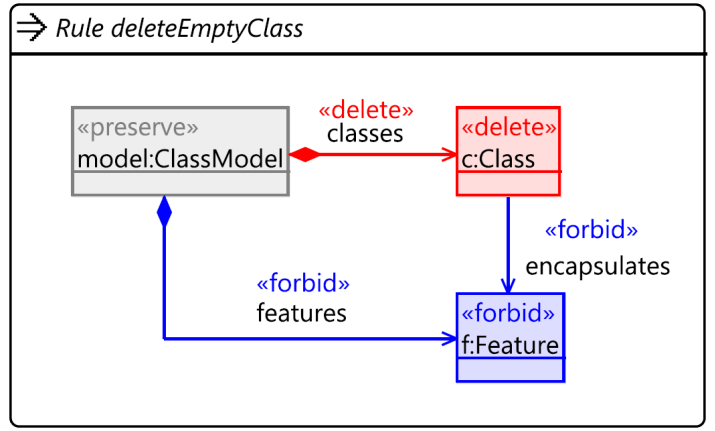
Korrekter und vollständiger Regelsatz CRA





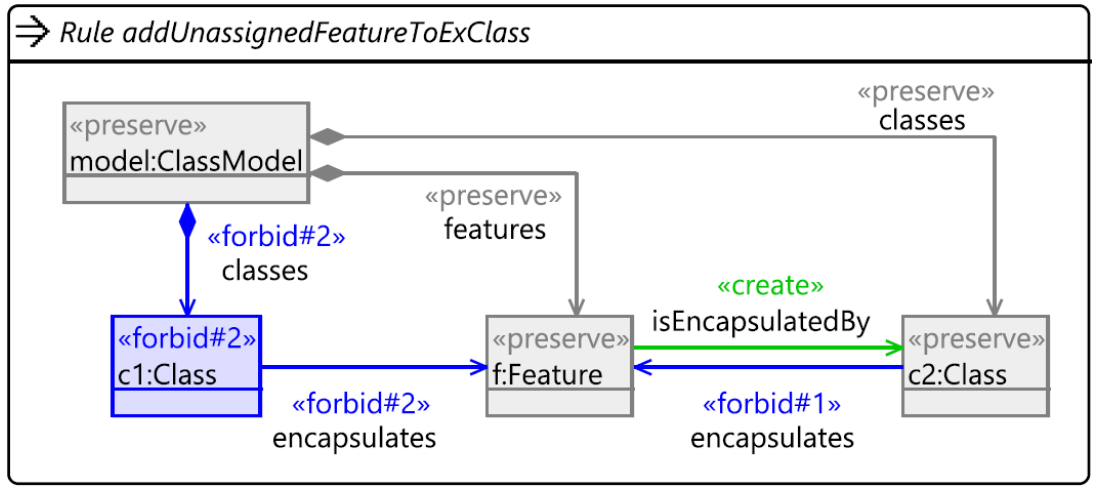


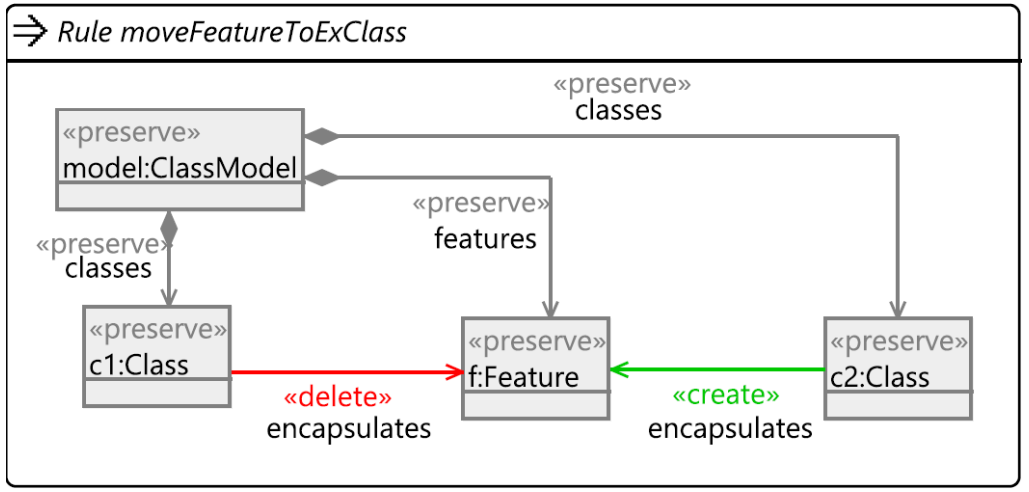


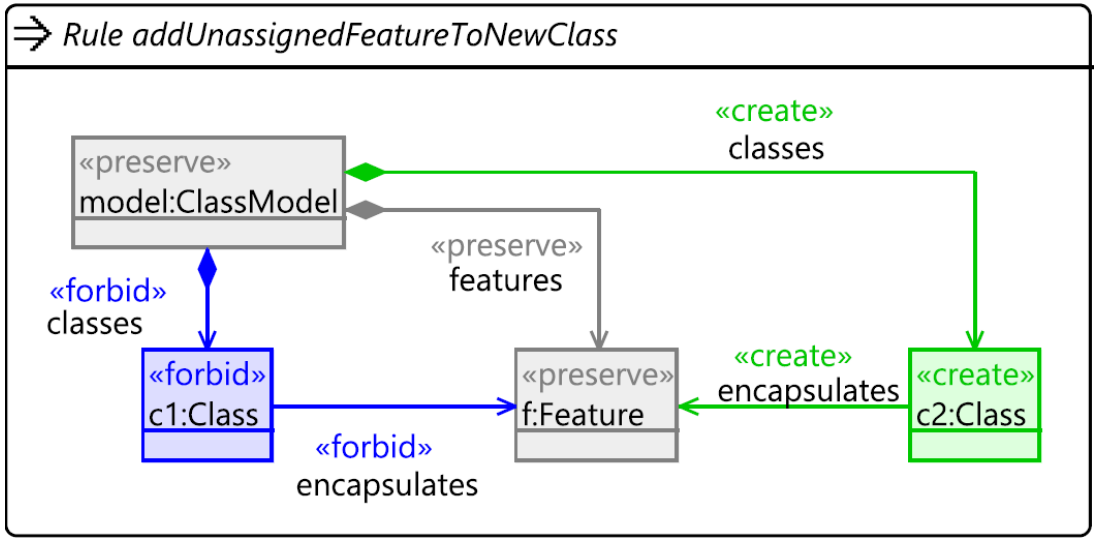


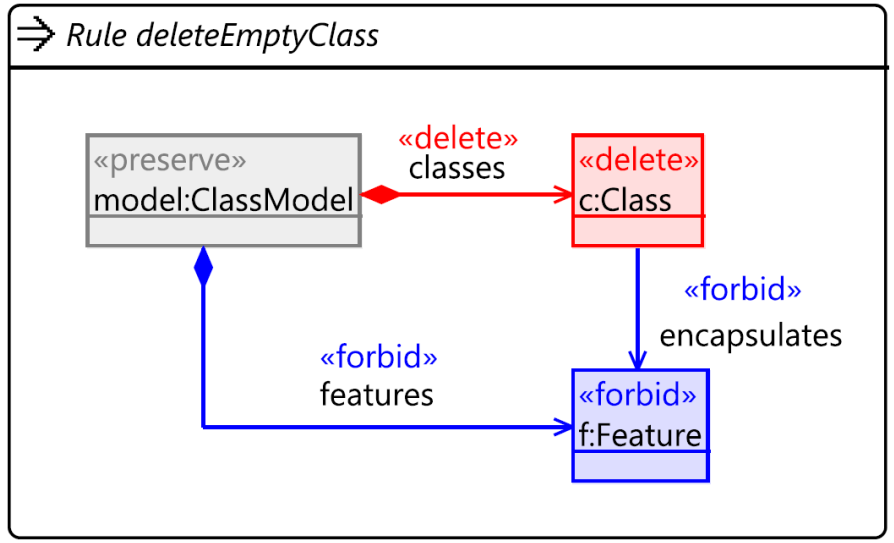
Quelle: [JKLT23]

Korrekter aber unvollständiger Regelsatz CRA



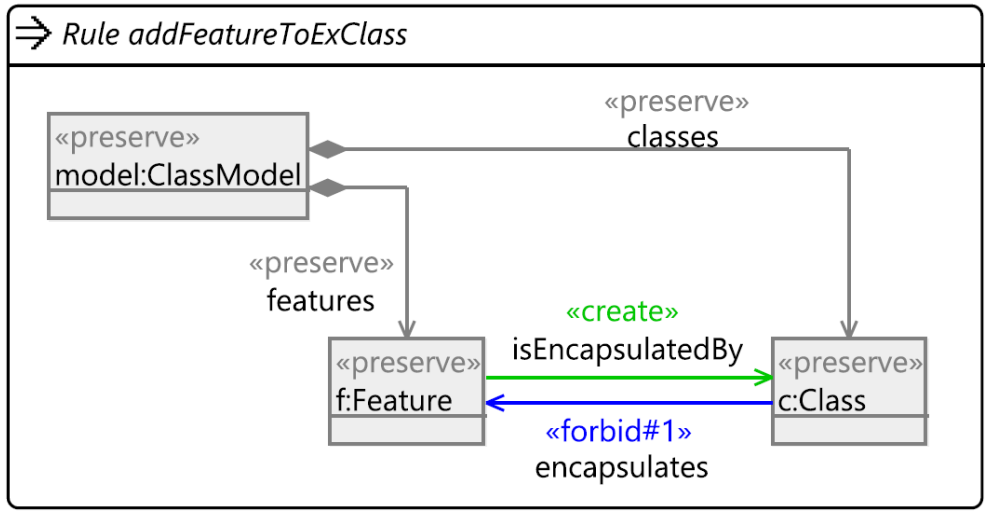


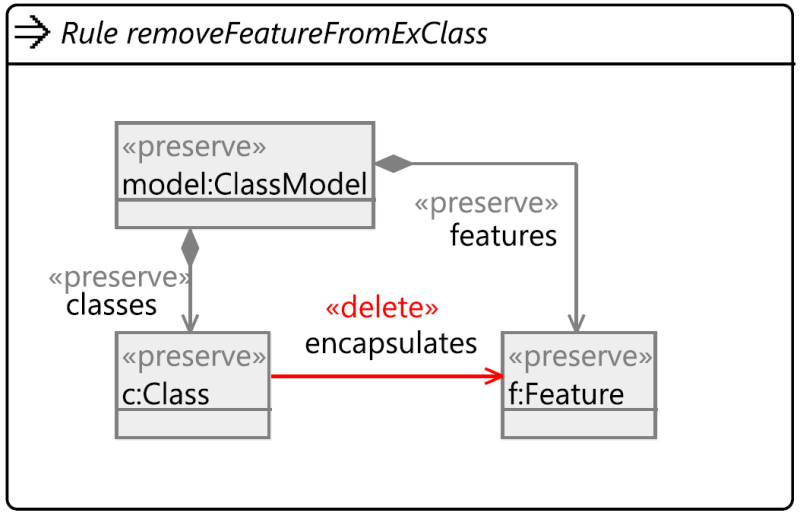


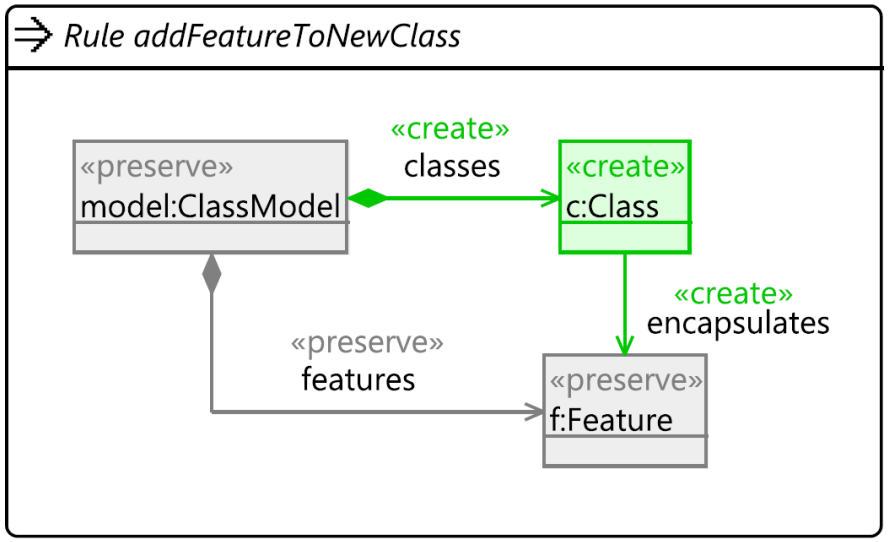


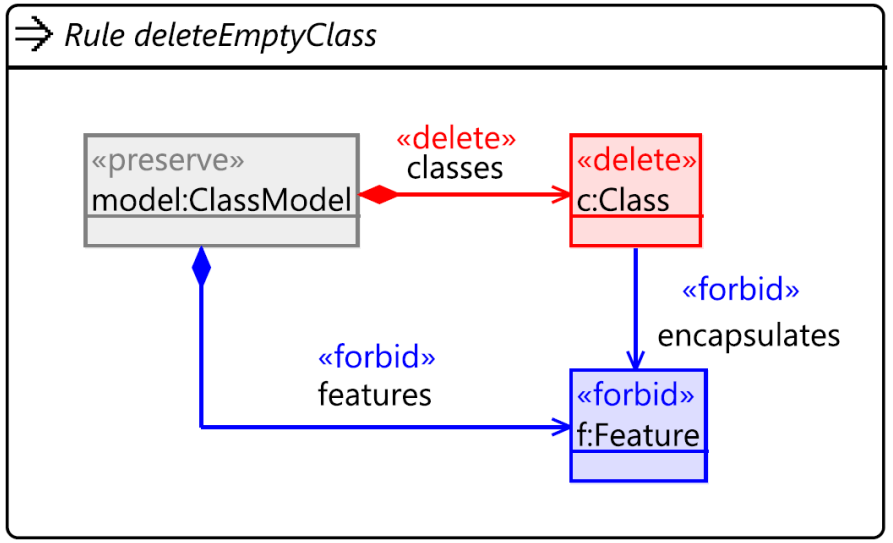
Quelle: [JKLT23]

Inkorrekter aber vollständiger Regelsatz CRA









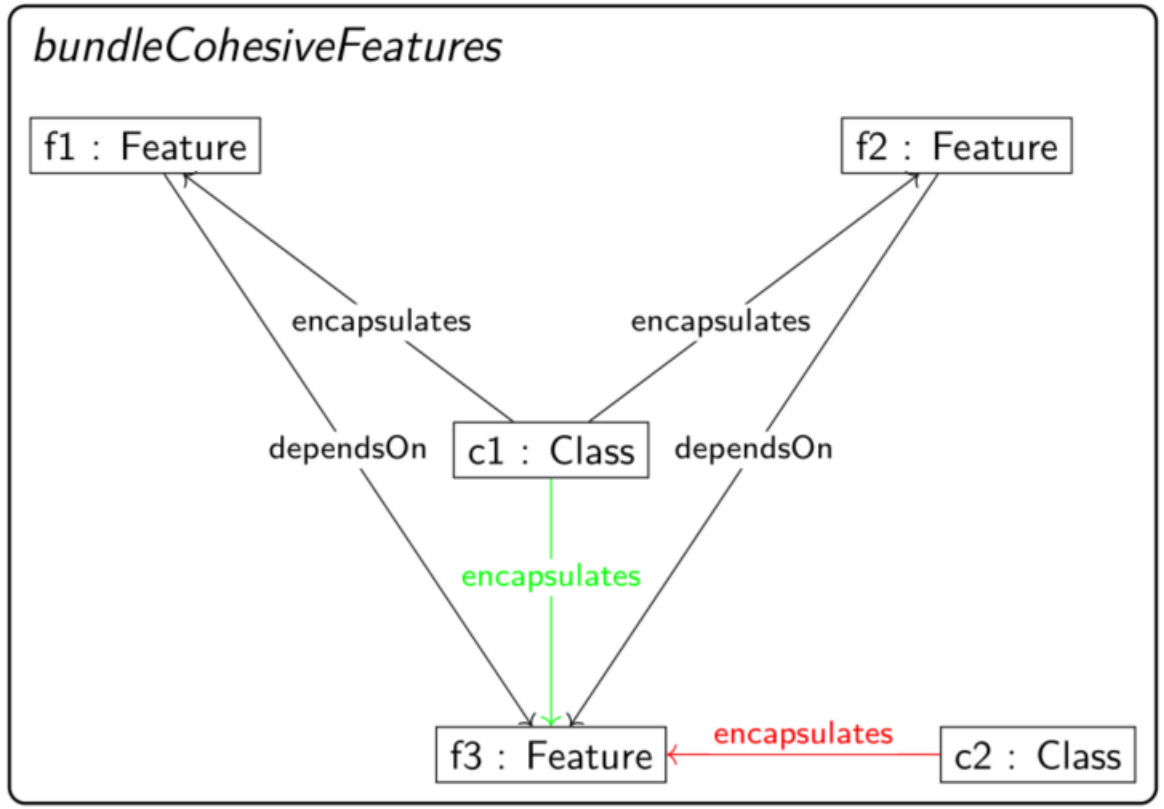
Quelle: [JKLT23]

Evaluation Eigenschaften von Graphmutation

Experiment berichtet in [JKLT23]:

- Drei verschiedene Optimierungsprobleme, wo jeweils ein Metamodell (Klassendiagramm) plus Nebenbedingungen den Suchraum von Graphen kodieren (inklusive CRA).
- Für jedes Problem verschiedene konkrete Probleminstanzen und drei verschiedene Sätze von Mutationsoperatoren: korrekt und vollständig (kv); korrekt und unvollständig (kuv); unkorrekt und vollständig (ukv)
- Einsatz von drei verschiedenen evolutionären Algorithmen
- Ergebnisse:
  - ukv schneidet klar am schlechtesten ab (findet nur Lösungen deutlich schlechterer Qualität); lediglich die Zeit pro Iteration ist gut
  - Der Vergleich zwischen kv und kuv schwankt mit Problem, konkreter Instanz und verwendetem Algorithmus. Tendenz: kuv ist effizienter (terminiert schneller) aber kv effektiver (findet Lösungen höherer Qualität).
  - Für kuv ist entscheidend, welcher Teil des Suchraums nicht mehr erreichbar ist.

Entwurf problemspezifischer Regeln



- Graphtransformationsregeln oder -programme können versuchen, problemspezifische Suchheuristiken zu implementieren.
- Beispiel: Regel *bundleCohesiveFeatures* für das CRA-Problem
  - Feature mit gemeinsamer Abhängigkeit wird in die gemeinsame Klasse eingeordnet.
  - Regelanwendung kann (muss aber nicht!) gleichzeitig Kohäsion erhöhen und Kopplung senken.

Zusammenfassung Graphmutationen

- Das Mutieren von Graphen kann durch Programme, die einen Graphen (in geringem Umfang) modifizieren, umgesetzt werden. Graphtransformationsregeln bieten eine Möglichkeit, solche Programme zu entwerfen
- Mutationen müssen für jedes Optimierungsproblem neu entworfen werden; für verschiedene Instanzen des gleichen Optimierungsproblems sind sie dann normalerweise einsetzbar.
- Für Eigenschaften wie Korrektheit und Vollständigkeit muss von Fall zu Fall argumentiert werden; durch die detaillierten Informationen im Graph ist der Entwurf von korrekten/vollständigen Mengen von Mutationen aber oft möglich.
- Eigenschaften wie Korrektheit und Vollständigkeit können entscheidenden Einfluss auf den Erfolg eines evolutionären Algorithmus haben.

Ausblick: Ein Basissatz von Mutationsregeln kann aus Klassendiagrammen generiert werden.

Rekombinationsoperatoren auf Graphen

Problemspezifische Ansätze:

- Domänenwissen und Semantik der Graphrepräsentation werden ausgenutzt.
- Für jedes Optimierungsproblem müssen neue Operatoren entworfen und implementiert werden.
- Auf problemspezifische Nebenbedingungen kann Rücksicht genommen werden.

Allgemeine Ansätze:

- Rekombinationsoperatoren, die auf jeder Art von Graph funktionieren, unabhängig von Semantik und Optimierungsproblem.
- Können über Optimierungsprobleme hinweg wiederverwendet werden.
- Nehmen keine Rücksicht auf problemspezifische Nebenbedingungen.

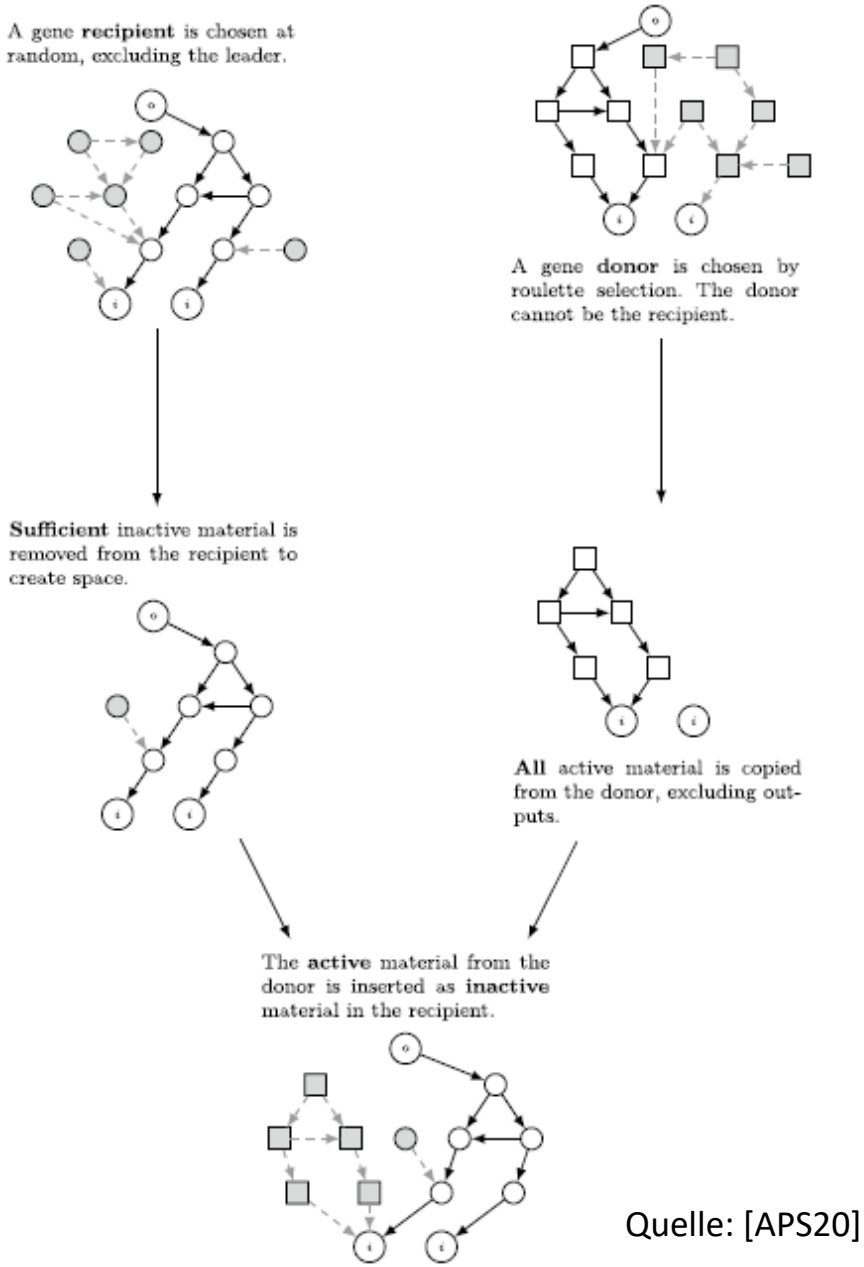
Problemspezifische Rekombination in [APS20]

Atkinson et al. schlagen in [APS20] einen Rekombinationsoperator für Graphen vor, den sie horizontal gene transfer (HGT) nennen:

- Erinnerung: In [APS20] kodieren Graphen Funktionen und ausgehend vom Output-Knoten gibt es in jedem Graphen einen Baum, der die kodierte Funktion als expression tree beschreibt (active component)
- Ein Empfänger wird unabhängig gleichverteilt aus der Population ausgewählt (ausgeschlossen das beste Individuum).
- Ein Donor wird so ausgewählt, dass Individuen mit hoher Fitness bevorzugt werden (Roulette Wheel Selection).
- Aus dem Empfänger werden neutrale Knoten (und angrenzende Kanten) gelöscht und stattdessen die active component des Donors eingefügt (so, dass die Anzahl der Knoten erhalten bleibt).

Beispiel HGT

- Semantik von Donor und Empfänger bleiben unverändert.
- Durch spätere Mutation kann das neue Material im Empfänger (das aus einem Individuum mit hoher Fitness stammt) „aktiviert“ werden.
- HGT kann so durchgeführt werden, dass die Nebenbedingungen eingehalten werden (iterativ Knoten löschen, deren ausgehende Kanten gelöscht wurden).
- HGT ist nur einsetzbar in Fällen, wo die Unterscheidung zwischen „aktiver“ und „neutraler“ Komponente Sinn ergibt.

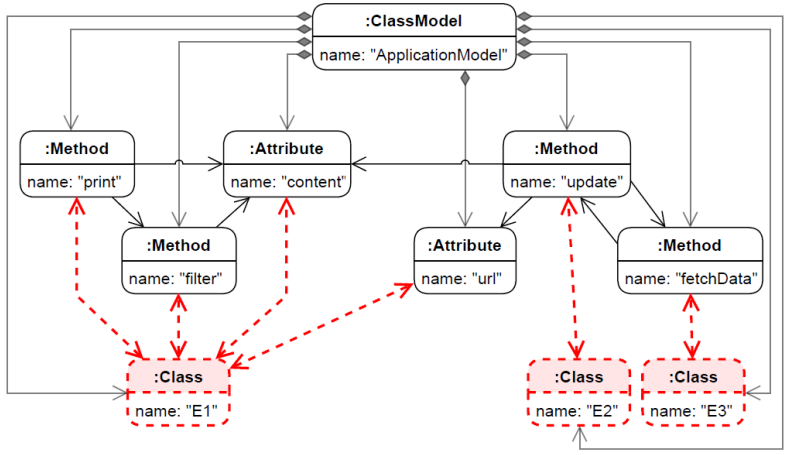


Quelle: [APS20]

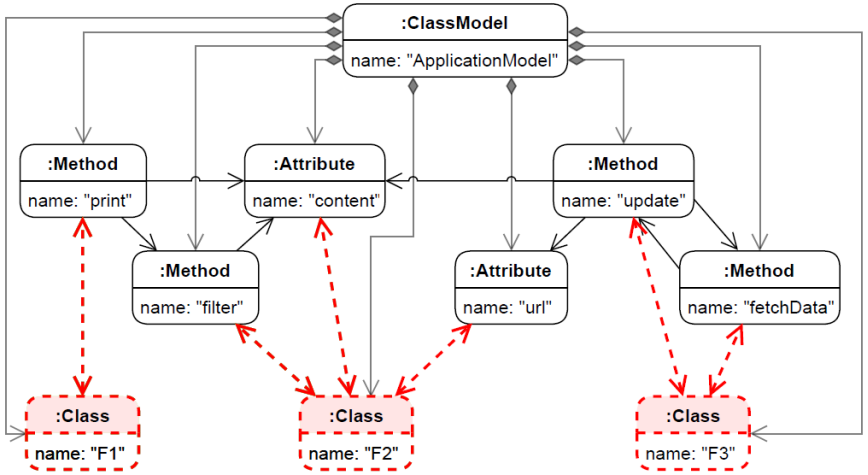
Allgemeine Ansätze für die Rekombination von Graphen

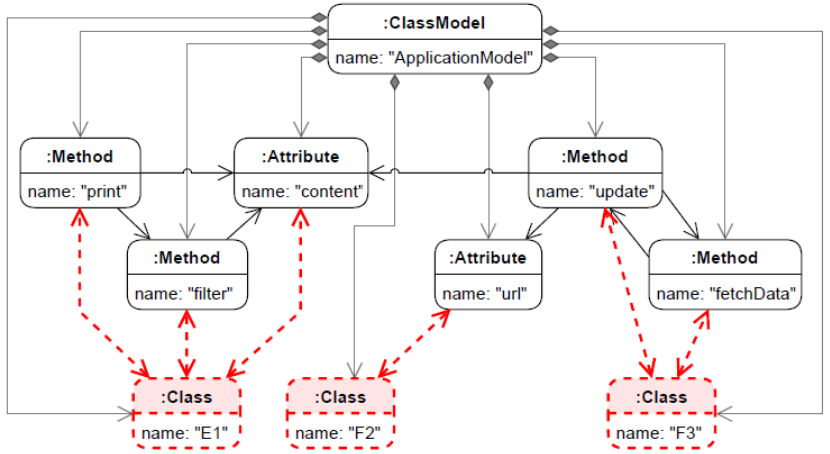
- Austausch bzw. Transfer von Untergraphen:
  - Unterschiedliche Ansätze in der Literatur: [GALW00, Nie03, MNR10]
  - Kernidee: ein Untergraph U_1 aus einem Graphen G_1 ersetzt einen Untergraphen U_2 in einem Graphen G_2 (und umgekehrt)
  - Details, wie die Untergraphen ausgewählt und in ihren neuen Graphen eingebunden werden, unterscheiden sich.
- Verallgemeinerung: Verklebung von Teilgraphen [KJT23]. Kernidee:
  - Zwei Graphen G_1 und G_2 werden in je zwei Teile G_i^1 und G_i^2 aufgeteilt ($i = 1,2$); diese Teile müssen nicht disjunkt sein.
  - Die Untergraphen G_1^1 und G_2^2 bzw. G_1^2 und G_2^1 werden jeweils über einem gemeinsamen Untergraphen $U \subseteq G_1^1 \cap G_1^2 \cap G_2^1 \cap G_2^2$ vereinigt (leichte technische Vereinfachung!). Es können also Knoten aus G_1 und G_2 identifiziert werden.

Beispiel Verklebung von Teilgraphen [KJT23]

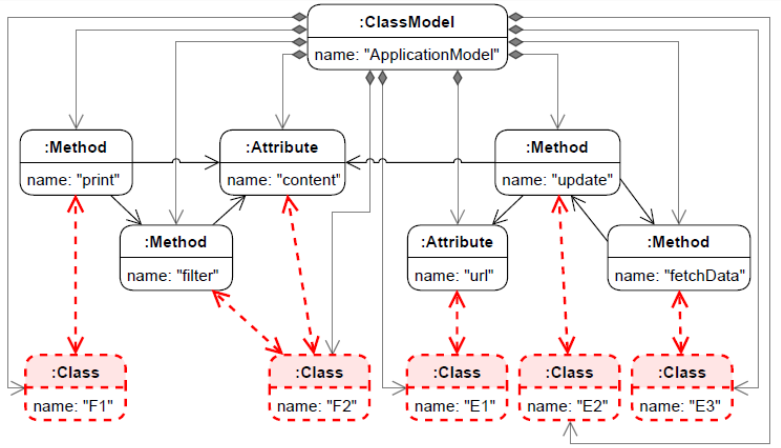


Elternmodelle





Nachkommen



Evaluationen von Rekombinationsoperatoren auf Graphen

- Für problemspezifische Rekombinationsoperatoren liegen häufig Experimente vor, in denen sie gut abschneiden (im Vergleich zu Algorithmen ohne Rekombinationsoperator); z.B. [APS20]
- Für allgemeine Ansätze ist die Lage gemischt; z.B. Experiment in [JKT23]:
  - Zusätzlicher Einsatz von Crossover verschlechtert die Effektivität des evolutionären Algorithmus (wegen der vielen eingeführten Verletzungen von Nebenbedingungen).
  - Mit zusätzlicher Reparatur ist evolutionäre Suche mit Crossover und Mutation effektiver als mit Mutation allein.

Variationsoperatoren für Vektoren natürlicher Zahlen

Mutation:

- Positionsweises zufällig gleichverteiltes Setzen auf neuen Wert wie bei Stringkodierung bereits gesehen (vor allem, wenn natürliche Zahlen in kardinaler Bedeutung verwendet werden)
- Positionsweise Addition einer zufälligen (kleinen) natürlichen Zahl (vor allem, wenn natürliche Zahlen in ordinaler Bedeutung verwendet)

Rekombination:

- k-Punkt und Uniform Crossover wie in der Stringkodierung bereits gesehen

Variationsoperatoren auf Vektoren reeller Zahlen

Suchraum im Folgenden:

- Vektoren $(v_1, \dots, v_n) \in [a_1, b_1] \times \cdots \times [a_n, b_n] \subseteq \mathbb{R}^n$
- In der Theorie gehen wir von reellen Einträgen aus; in Implementierung natürlich floating point numbers.
- Die einzelnen Einträge werden oft Variablen genannt.

Mutation:

- Positionsweises zufällig gleichverteiltes Setzen auf neuen Wert
- Positionsweise Addition einer zufälligen (kleinen) Zahl (meistens)

Rekombination:

- k-Punkt oder uniform crossover (genannt diskrete Rekombination)
- Durchschnitts- bzw. arithmetische Rekombination (jedes Allel nimmt Wert zwischen entsprechenden Allelen der Eltern ein)
- Mischrekombination (jedes Allel nimmt Wert in der Nähe eines der Elternwerte an)

Mutation von Vektoren reeller Zahlen

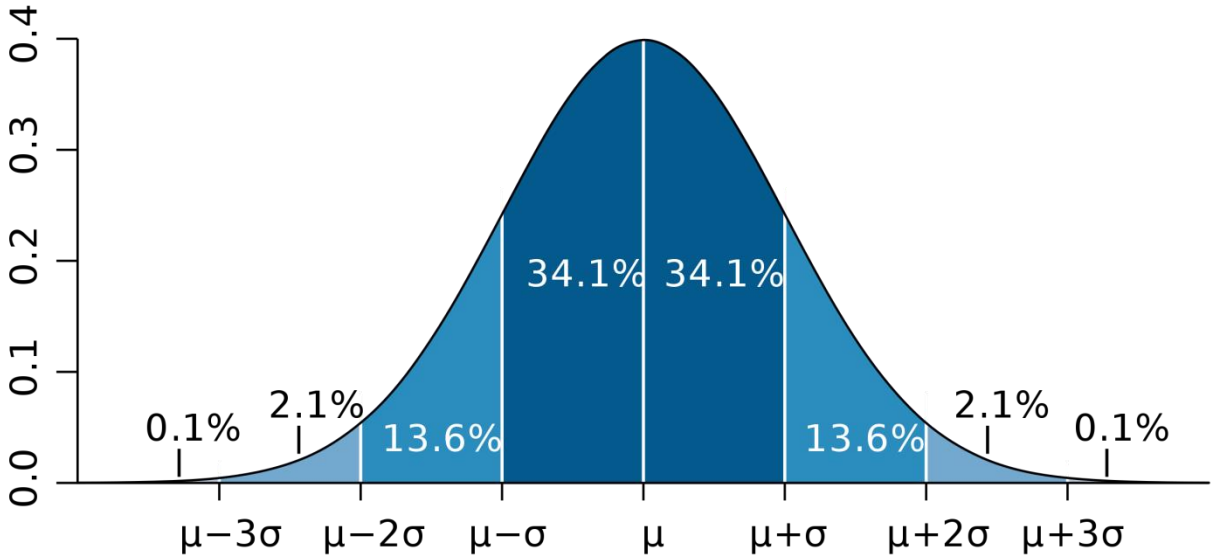
Verbreitester Mutationsoperator:

- Gegeben Vektor $(v_1, \ldots, v_n)$ wird zu jeder Variable v_i ein um 0 normalverteilt gesampelter Wert addiert. Notation:

$$(v_1, \ldots, v_n) \mapsto (v_1', \ldots, v_n'),$$
$$v_i' = v_i + N(0, \sigma)$$

- Die Standardabweichung σ wird hier der entscheidende Parameter, der das Ausmaß der Änderung kontrolliert; Name: mutation step size. Mutationswahrscheinlichkeit p_m wird auf 1 gesetzt (jedes Chromosom mutiert).
- Falls nötig wird der Wert v_i' auf die (oder leicht ober- bzw. unterhalb der) Intervalgrenze von $[a_i, b_i]$ gesetzt.

Normalverteilung



Wahrscheinlichkeitsdichtefunktion der Normalverteilung; Quelle: Ainali, Standard deviation, CC BY-SA 3.0, via Wikimedia Commons

- Dichtefunktion

$$\frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

- Gut zwei Drittel der Werte sind innerhalb von einer Standardabweichung um den Mittelwert verteilt.

Eigenschaften Mutation Vektoren reeller Zahlen

- Korrektheit: Mutation berechnet Vektor reeller Zahlen, sodass alle Einträge innerhalb der ggf. vorgegebenen Intervalle liegen.
- Vollständigkeit:
  - Jeder Vektor $(x_1, \dots, x_n)$ ist von jedem Vektor $(y_1, \dots, y_n)$ aus per Mutation erreichbar (mit extrem geringer Wahrscheinlichkeit sogar in einem Schritt).
  - Praktisch führt eine Mutation meistens jedoch eine kleine, lokale Änderung durch. ⇒ Zur Erkundung neuer Gebiete des Suchraums wird große mutation step size σ oder ein Rekombinationsoperator gebraucht.

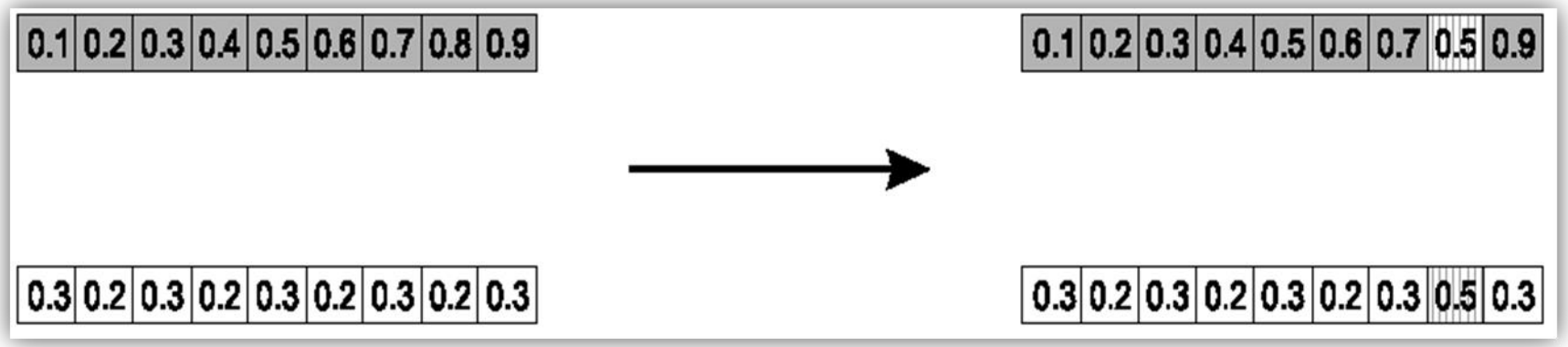
Arithmetische Crossoveroperatoren

- Grundidee: Werte der Kindchromosomen liegen „zwischen“ den Werten der Elternchromosomen.
- Berechnungsvorschrift: Aus Elternchromosomen $(x_1, \dots, x_n)$ und $(y_1, \dots, y_n)$ werden Einträge z_i als
$$z_i = \alpha x_i + (1 - \alpha) y_i$$
für $0 \leq \alpha \leq 1$ berechnet.
- Unterschiedliche Operatoren berechnen unterschiedliche Einträge auf diese Weise.
- Parameter α kann
  - konstant sein;
  - auf Grundlage irgendeiner Regel variieren (z.B. Iterationszahl);
  - jedes Mal zufällig neu bestimmt werden.

Arithmetischer Crossover: Single arithmetic crossover

Gegegeben sind Eltern $(x_1, \dots, x_n)$ und $(y_1, \dots, y_n)$. Ablauf:

1. Eine Position $1 \leq k \leq n$ wird zufällig gleichverteilt gewählt.
2. „Mischen“ an Position k: Der erste Nachkomme wird berechnet als
$$z_1 = (x_1, \dots, \alpha y_k + (1-\alpha)x_k, x_{k+1}, \dots, x_n).$$
3. Der zweite Nachkomme wird mit umgekehrten Rollen berechnet.

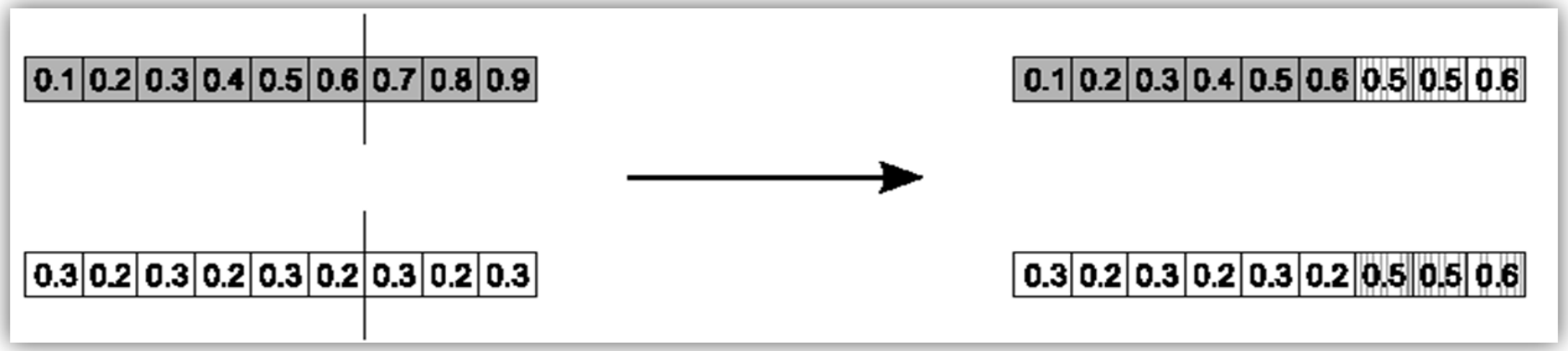


Beispiel für single arithmetic crossover mit $k = 8$ und $\alpha = 0.5$.

Arithmetischer Crossover: Simple arithmetic crossover

Gegegeben sind Eltern $(x_1, \dots, x_n)$ und $(y_1, \dots, y_n)$. Ablauf:

1. Eine Position $0 \leq k \leq n$ wird zufällig gleichverteilt gewählt.
2. „Mischen“ ab Position $k + 1$: Der erste Nachkomme wird berechnet als
$$z_1 = (x_1, \dots, x_k, \alpha y_{k+1} + (1 - \alpha)x_{k+1}, \dots, \alpha y_n + (1 - \alpha)x_n).$$
3. Der zweite Nachkomme wird mit umgekehrten Rollen berechnet.

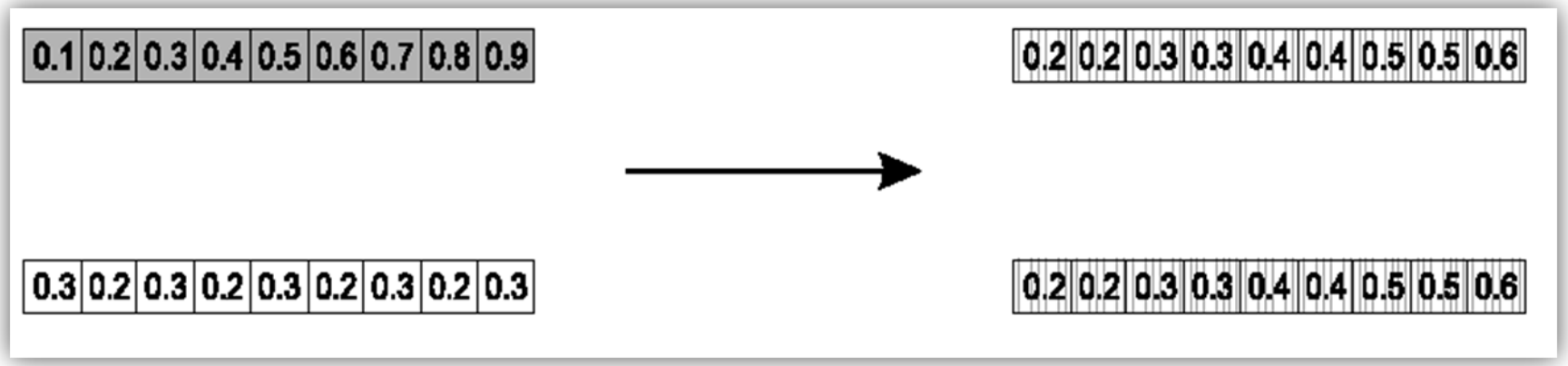


Beispiel für simple arithmetic crossover mit $k = 6$ und $\alpha = 0.5$.

Arithmetischer Crossover: Whole arithmetic crossover

Die am häufigsten eingesetzte Variante. Gegegeben sind Eltern $(x_1, \dots, x_n)$ und $(y_1, \dots, y_n)$. Ablauf:

1. Mischen an jeder Position: Der erste Nachkomme wird berechnet als
$$z_1 = (\alpha y_1 + (1-\alpha)x_1, \dots, \alpha y_n + (1-\alpha)x_n).$$
2. Der zweite Nachkomme wird mit umgekehrten Rollen berechnet (oder nur ein Nachkomme).



Beispiel für whole arithmetic crossover mit $\alpha = 0.5$.

Mischrekombination (blend crossover – BLX-α)

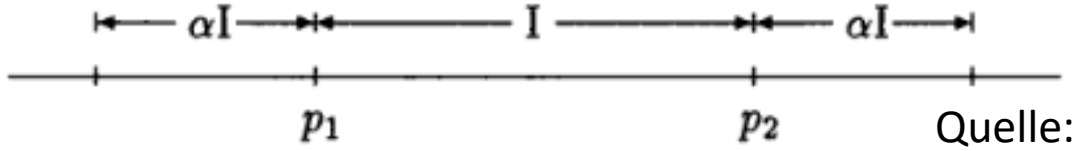
Gegegeben sind Eltern $(x_1, \ldots, x_n)$ und $(y_1, \ldots, y_n)$.

- Für jeden Eintrag $1 \leq i \leq n$ setze $I_i = |x_i - y_i|$.
- Jeder Eintrag z_i des Kindes wird durch zufällig gleichverteilte Wahl aus dem Interval

$$[\min(x_i, y_i) - \alpha I_i, \max(x_i, y_i) + \alpha I_i]$$

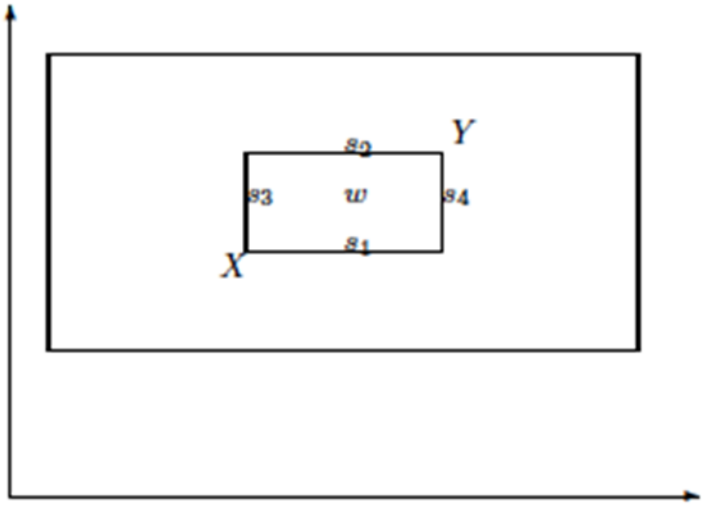
bestimmt, wobei $0 \leq \alpha \leq 1$.

- Meist wird $\alpha = 0.5$ verwendet.



Quelle: [ES92]

Eigenschaften Crossover auf Vektoren reeller Zahlen



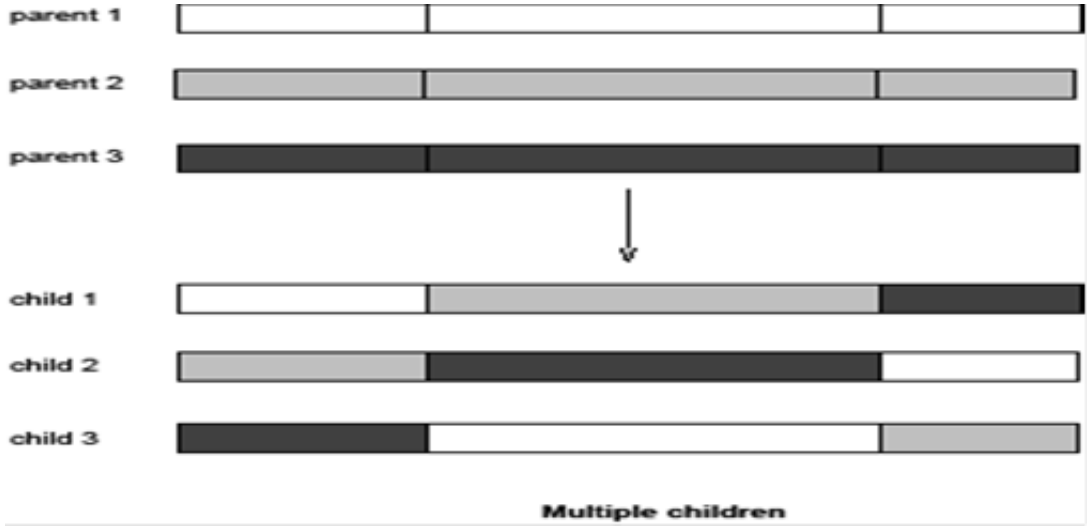
Schematische Darstellung (in zwei Variablen), welche Nachkommen von den Eltern X und Y aus erreichbar sind.

X und Y sind (zweidimensionale) Elternchromosomen.

- Single arithmetic crossover: Für $\alpha = 0.5$ sind s_1, s_2, s_3, s_4 als Nachkommen berechenbar. Für andere Wahlen von α verschieben sich die berechenbaren Nachkommen entsprechend entlang des inneren Rahmens.
- Whole arithmetic crossover: Für $\alpha = 0.5$ wird w als Nachkomme berechnet. Für andere Werte von α verschiebt sich das Ergebnis entlang der Geraden durch X und Y.
- BLX-α: Zufällig gleichverteilte Wahl innerhalb des äußeren Rahmens (hier mit $\alpha = 1$).

Crossover mit mehr als zwei Eltern

- Vereinzelt wurde Crossover auch auf mehr als zwei Eltern angewendet – am häufigsten auf Vektoren reeller Zahlen.
- Ansätze:
  - Verallgemeinerung von 1-Punkt Crossover
  - Verallgemeinerung von arithmetic crossover



Literatur zu Permutationen

- [GL85] D. E. Goldberg, R. Lingle Jr.: Alleles, Loci, and the Traveling Salesman Problem. ICGA 1985: 154–159
- [OSH87] I. M. Oliver, D. J. Smith, J. R. C. Holland: A Study of Permutation Crossover Operators on the Traveling Salesman Problem. ICGA 1987: 224–230
- [SMcDMWW91] T. Starkweather, S. McDaniel, K. E. Mathias, L. D. Whitley, C. Whitley: A Comparison of Genetic Sequencing Operators. ICGA 1991: 69 – 76
- [Potvin96] J.-Y. Potvin: Genetic algorithms for the traveling salesman problem. Ann. Oper. Res. 63(3): 337–370 (1996)
- [LKMID99] P. Larrañaga, C. M. H. Kuijpers, R. H. Murga, I. Inza, S. Dizdarevic: Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators. Artif. Intell. Rev. 13(2): 129–170 (1999)
- [Whitley00] D. Whitley: Permutations. In T. Bäck, D. B. Fogel, Z. Michalewicz: Evolutionary Computation 1. Basic Algorithms and Operators. Bristol 2000: 274–284.

Literatur zu Graphen

- [GALW00] A. Globus, S. Atsatt, J. Lawton, T. Wipke, JavaGenes: Evolving Graphs with Crossover, Tech. Rep. NAS-00-018, NASA (2000)
- [Nie03] J. Niehaus: Graphbasierte Genetische Programmierung. Dissertation. Technische Universität Dortmund (2003)
- [MNR10] P. Machado, H. Nunes, J. Romero:´Graph-Based Evolution of Visual Languages. EvoApplications (2) 2010: 271–280
- [APS20] T. Atkinson, D. Plump, S. Stepney: Horizontal gene transfer for recombining graphs. Genet. Program. Evolvable Mach. 21(3): 321–347 (2020)
- [JKT23] Stefan John, Jens Kosiol, Gabriele Taentzer: Towards a configurable crossover operator for model-driven optimization. MoDELS (Companion) 2022: 388–395
- [JKLT23] S. John, J. Kosiol, L. Lambers, G. Taentzer: A graph-based framework for model-driven optimization facilitating impact analysis of mutation operator properties. Softw. Syst. Model. 22(4): 1281–1318 (2023)
- [KJT23] J. Kosiol, S. John, G. Taentzer: A generic construction for crossovers of graph-like structures and its realization in the Eclipse Modeling Framework. J. Log. Algebraic Methods Program. 136 (2024)

Literatur zu Vektoren reeller Zahlen

- [ES92] L. J. Eshelman, J. D. Schaffer: Real-Coded Genetic Algorithms and Interval-Schemata. FOGA 1992: 187–202