Aufgabe 3: Eisbudendilemma

Teilnahme-ID: 01143

Bearbeiter/-in dieser Aufgabe:
Nils Weißer

19. April 2021

Inhaltsverzeichnis

1 Lösungsidee

Bei der Aufgabe „Eisbudendilemmma“ handelt es sich um ein Optimierungsproblem. Der Lösungsraum ist jede Kombination, die man mit den drei Eisbuden um den See herum, finden kann. Das Ziel ist es eine Bewertungsfunktion zu finden, die eine Kombination von Eisbuden beurteilen kann. Die Schwierigkeit liegt darin, dass eine Kombination von Eisbuden alleinstehend keine Bewertung erhalten kann, sondern sie nur im Vergleich mit einer anderen Kombination bewertet werden kann.

1.1 Brute-Force

Die einfachste Methode das Problem zu lösen, wäre die Brute-Force-Suche. Dabei wird jede Kombination der Eisbuden, mit allen anderen Kombinationen der Eisbuden nach den vorgegebenen Regeln verglichen.

Wenn eine andere Kombination bei einer Abstimmung der Dorfbewohner mehr Stimmen kriegen würde, ist die untersuchte Kombination nicht stabil.
Mit dieser Methode würde man immer ein optimales Ergebnis erhalten. Das Problem der Brute-Force-Suche ist die Anzahl der Operationen. Die Anzahl der Kombinationen, die die Eisbuden haben können, kann mit der Formel für die Kombination ohne Wiederholung berechnet, da keine Eisbuden an der gleichen Position sein können. Wenn n die Anzahl der möglichen Positionen (der Umfang) ist und k die Anzahl der Eisbuden (in diesem Fall 3) ist die Formel also:

$$\frac{n!}{(n-k)! \cdot k!}$$

Nun muss jede Kombination noch mit allen anderen Kombinationen verglichen werden, also entspricht die Anzahl der Vergleiche:

$$(\frac{n!}{(n-k)! \cdot k!})^2 - \frac{n!}{(n-k)! \cdot k!} \text{ , bzw. für k=3 } (\frac{n!}{(n-3)! \cdot 6})^2 - \frac{n!}{(n-3)! \cdot 6}$$

Das bedeutet, dass die bei steigendem n (d.h. steigendem Umfang) die Anzahl der Operationen zunächst faktoriell und daraufhin quadratisch steigt. Bei einem Umfang von 200 wäre die Anzahl der Vergleichsoperation bereits

$$(\frac{200!}{(200-3)! \cdot 6})^2 - \frac{200!}{(200-3)! \cdot 6} = 1,725 \cdot 10^{12}$$

Somit ist die Brute-Force-Methode bei größeren Umfängen nicht mehr realisierbar.

1.2 Genetischer Algorithmus

Um die Laufzeitkomplexität des Problems herabzusetzen, wird ein genetischer Algorithmus entwickelt. Doch was ist ein genetischer Algorithmus überhaupt?
Das Prinzip des genetischen Algorithmus beruht auf der, aus der Evolutionstheorie stammenden, natürlichen Selektion. Natürliche Selektion bedeutet grundlegend, dass Individuen einer Population eine höhere Fortpflanzungsrate haben, wenn sie stärker sind. Nach diesem Prinzip soll mit einem genetischen Algorithmus die „stärkste“ Lösungsmöglichkeit, bzw. im Optimalfall die richtige Lösung gefunden werden, indem sich von Generation zu Generation immer bessere Lösungsmöglichkeiten bilden.

Der genetische Algorithmus besteht aus 5 Schritten.
1. Initialisierung:
Zunächst wird die Anfangspopulation (die erste Generation) zufällig bestimmt.
2. Selektion:
Es wird eine Auswahl and Individuen getroffen, die sich in der Anfangspopulation befinden, aus denen die nächste Generation gebildet wird. Dabei werden „stärkere“ Individuen eher gewählt, als „schwächere“. Um die Individuen auszuwählen, muss eine Fitness-Funktion definiert werden, die bestimmt, wie „gut“ die Lösungskombination ist.
3. Rekombination:
Die zuvor ausgewählten Individuen werden zufällig miteinander rekombiniert, sodass neue Individuen mit den Attributen der „Eltern“ entstehen.
4. Mutation:
Die neuen Individuen mutieren zufällig, das heißt sie können neue Attribute kriegen, die nicht von der vorherigen Generation stammen.
5. Termination:
Die Schritte 2, 3 und 4 werden so lange ausgeführt, bis eine Terminationsbedingung erfüllt wird. Das könnte zum Beispiel sein, dass die letzten Generationen sich kaum verändert haben oder eine Maximalanzahl an Generationen erreicht wurde.

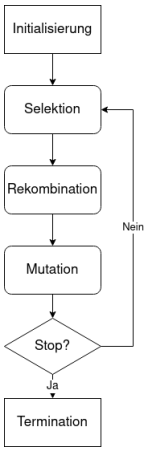

Abb. 1: Übersicht über die Abfolge des genetischen Algorithmus

1.3 Anwendung auf Problemstellung

Nun müssen die beschriebenen Schritte auf die Problemstellung angepasst werden, sodass das Problem durch einen genetischen Algorithmus gelöst werden kann.

1.3.1 Initialisierung

In diesem Schritt muss die erste Generation generiert werden.
Hier gibt es einige Möglichkeiten die erste Generation nicht komplett zufällig zu erstellen, sondern einige Heuristiken oder Regeln aufzustellen, sodass sie bereits in die richtige Richtung der Lösung geht. Der Einfachheit halber werden jedoch rein zufällige Kombinationen verwendet.
Im Vorhinein muss jedoch noch bestimmt werden, wie viele Lösungskombinationen sich in einer Generation befinden. Diese Variable wird p_a genannt.

1.3.2 Selektion

In diesem Schritt müssen aus der vorherigen Generation Lösungskombinationen ausgewählt werden, die für die nächste Generation verwendet werden können.
Um die besten Lösungskombinationen auszuwählen muss zunächst ein Maß gefunden werden, dass bestimmt wie gut, bzw. wie nah an der Lösung, eine Lösungskombination ist. Dafür wird eine Fitness-Funktion definiert.
Jede Lösungskombination hat zunächst p_a, also die Anzahl der Kombinationen in der Generation, als Fitness-Wert. Der Fitness-Wert wird mit f bezeichnet. Dann wird jede Lösungskombination mit allen anderen aus der aktuellen Generation verglichen. Wenn eine der Kombinationen gegen die andere, nach den in der Aufgabenstellung genannten Regeln, gewinnen würde, wird der Fitness-Wert dieser Kombination um 1 verringert. Dafür müssen zunächst die Abstände aller Häuser zu den Standorten der Kombination berechnet werden. Dabei muss auf den Kreis als nicht-lineare Struktur geachtet werden. Die Lösungskombinationen, die am Ende einen geringen Fitness-Wert haben, sind also „stärker“ im Vergleich zu den anderen Lösungskombinationen der Generation und somit im Optimalfall näher an der Lösung.
Nach diesem Fitness-Wert f werden nun die Kombinationen für die nächste Generation ermittelt. Dafür wird jede Kombination zunächst $p_a - f$ mal in den „Pool“ geworfen. Daraufhin werden p_a Kombinationen aus diesem „Pool“ zufällig ausgewählt. Das führt dazu, dass eine Kombination mit einem niedrigeren Fitness-Wert eher ausgewählt wird, als eine Kombination mit einem höheren Fitness-Wert.

1.3.3 Rekombination

Zunächst muss bestimmt werden, wie wahrscheinlich es ist, dass eine Lösungskombination rekombiniert wird. Dafür wird eine Variable p_r definiert, die diese Wahrscheinlichkeit angibt.
Wenn eine Kombination dann rekombiniert werden soll, muss zudem noch zufällig bestimmt werden, welcher Teil der Kombination bestehen bleiben soll und wie groß dieser Teil ist. Außerdem muss noch eine andere Kombination zufällig ausgewählt werden, von der Eisbudenstandorte übernommen werden.

1.3.4 Mutation

Auch für die Mutation muss eine Wahrscheinlichkeit bestimmt werden, wie häufig ein Eisbudenstandort durch einen anderen ersetzt wird. Diese Wahrscheinlichkeit wird mit p_m bezeichnet.
Wenn ein Eisbudenstandort mutieren soll, wird zufällig ein anderer Eisbudenstandort bestimmt, mit dem der vorherige Standort ersetzt wird. Diese Mutation ist notwendig, da anderweitg nur Standorte für die Eisbuden in Frage kommen, die bereits in der ersten Generation auftraten.

1.3.5 Termination

Zuletzt muss noch eine Terminationsbedingung definiert werden.
Es werden für jede Generation die besten Lösungskombinationen, also diejenigen, die den niedrigsten Fitness-Wert haben, gespeichert. Wenn nun eine Lösungskombination zu einem Anteil von p_t der Generationen in den besten Lösungskombinationen vorkommt, wird der Algorithmus beendet. Außerdem muss eine Mindestanzahl an Generationen gen_{min} bestimmt werden, sodass die Terminationsbedingung nicht sofort eintrifft.
Da dieser Anteil p_t nicht immer erreicht wird, muss zudem noch eine Maximalanzahl an Generationen gen_{max} definiert werden. Somit ist die Terminiertheit des Verfahrens gegeben.

1.4 Auswahl der besten Kombinationen

Nachdem der Algorithmus terminiert wurde, muss bestimmt werden, ob mögliche stabile Standortkombinationen gefunden werden konnten und wenn ja, welche diese sind.
Dafür werden p_b Lösungskombinationen ausgewählt nach dem Kriterium, welche Kombinationen am häufigsten den niedrigsten Fitness-Wert einer Generation hatten. Diese Kombinationen werden erneut als Generation betrachtet und nach den Kriterien der „Selektion “ verglichen. Dabei wird für jede Kombination erneut ein Fitness-Wert bestimmt. Wenn alle Kombinationen mindestens gegen eine anderer verlieren, kann keine stabile Eisbudenstandortkombination gefunden werden. Wenn mindestens eine Kombination gegen keine andere verliert, ist sie möglicherweise eine stabile Standortkombination.

1.5 Bestimmung der Variablen

Die zuvor genannten Variablen müssen bestimmt werden, sodass das Ergebnis des Algorithmus möglichst optimal ist. Dafür werden zunächst mit einem Brute-Force-Algorithmus Beispieldaten gelöst, die noch lösbar sind. Daraufhin wird der Algorithmus mit verschiedenen Werten für die Variablen ausgeführt. Anschließend wird das Ergebnis mit dem korrekten, berechneten Ergebnis verglichen. Daran kann erkannt werden, bei welchen Werten die Ergebnisse am ehesten stimmen. Dann können die Werte weiter eingegrenzt werden und der Vorgang kann erneut durchgeführt werden, bis keine Optimierung mehr zu erkennen ist. Die folgenden Werte wurden nach dieser Methode für die Variablen bestimmmt:
$p_a = 55$
$p_r = 0,85$
$p_m = 0,085$
$p_t = 0.06$
$p_b = 20$
$gen_{min} = 15$
$gen_{max} = 1000$

1.6 Theoretische Analyse

Eine exakte Laufzeitanalyse zu ermitteln stellt sich als schwierig heraus, da die Anzahl der Generationen sehr stark auf Grund des Zufallsfaktors schwanken kann. Dennoch kann eine grobe Abschätzung getroffen werden.
Die Initialisierung hat eine Laufzeit von $\mathcal{O}(3 \cdot p_a)$, da für jede Kombination drei Standorte gefunden werden müssen.
Um die Laufzeit der Selektion zu berechnen, muss zunächst die Laufzeit der Fitness-Funktion bestimmt werden. In der Fitness-Funktion wird jede Lösungskombination aus der aktuellen Generation mit allen anderen verglichen. Daher ist die Laufzeit $\mathcal{O}(p_a \cdot (p_a - 1))$, bzw. $\mathcal{O}(p_a^2 - p_a)$. Nun müssen erneut p_a Kombinationen ausgewählt werden. Die Laufzeit dafür ist dementsprechend $\mathcal{O}(p_a)$.
Bei der Rekombination muss über alle Lösungskombinationen der Generation einmal iteriert werden.

Hierfür ist die Laufzeit also ebenso $\mathcal{O}(p_a)$.
Für die Mutation muss einmal über alle Lösungskombinationen und dann noch über jeden Eisbudenstandort der Kombinationen iteriert werden. Bei drei Eisbudenstandorten pro Kombination ist die Laufzeit also $\mathcal{O}(3 \cdot p_a)$. Für die Terminationsbedingung muss für jede beste Kombination der aktuellen Generation ermittelt werden, ob die Terminationsbedingung eintrifft. Da die Anzahl der besten Kombinationen stark schwanken kann und dieser Teil nicht sehr laufzeitaufwändig ist, wird er in der Berechnung weggelassen. Die Gesamtlaufzeit des Algorithmus, wenn G der Anzahl der Generationen entspricht, ist demnach:

$$\mathcal{O}(3 \cdot p_a + G \cdot (p_a^2 - p_a + p_a + p_a + 3 \cdot p_a)) = \mathcal{O}(3 \cdot p_a + G \cdot (p_a^2 + 4 \cdot p_a))$$

Nun kann noch der berechnete Wert $p_a = 55$ eingesetzt werden.

$$\mathcal{O}(165 + 3285 \cdot G)$$

An dieser Berechnung kann erkannt werden, dass die Laufzeit nicht auf den eingegeben Daten basiert, sondern lediglich auf der Anzahl der Generationen. Dennoch trifft die Terminationsbedingung in der Regel schneller ein, wenn der Umfang der Daten geringer ist.
Die Laufzeit von $\mathcal{O}(3285165)$ kann nicht überschritten werden, da $gen_{max} = 1000$ definiert ist. Im Vergleich zu der zuvor berechneten Laufzeit für den Bruteforce-Algorithmus ist die Laufzeit des genetischen Algorithmus nur ein Bruchteil dieser.

1.7 Probleme des genetische Algorithmus

Während die Laufzeit der eindeutige Vorteil des genetische Algorithmus gegenüber exakten Algorithmen ist, hat er dennoch einige Probleme.
Dadurch, dass dieser Algorithmus nicht exakt ist, kann es vorkommen, dass das Ergebnis nicht korrekt ist. Vor allem bei außergewöhnlichen Daten kommt es vermehrt dazu.
Zudem werden im Gegensatz zu einem exakten Algorithmus, auch wenn eine korrekte Lösung gefunden wurde, nicht immer alle möglichen Lösungen gefunden.
Ein weiteres Problem ist, dass der Algorithmus nicht determiniert ist. Wenn man ihn mehrfach ausführt ist nicht sichergestellt, dass das gleiche Ergebnis gefunden wird.

2 Umsetzung

Der Algorithmus wurde in Python implementiert.
In der Funtkion `initial_population()` wird die erste Generation initialisiert. Mit einer for-Schleife mit p_a Iterationen, werden die korrekte Anzahl an Kombinationen aus drei Eisbudenstandorten ermittelt. Zur zufälligen Auwahl wird die Python-Library „random“ verwendet. Zurückgegeben wird die erste Generation.
Diese Funktion wird beim ersten Aufruf der Funktion `ga()` mit aufgerufen. Die Funktion `ga()` hat als Übergabeparameter die aktuelle Generation und eine Variable, die die Anzahl der Generationen zählt. In der Funktion werden zunächst die Fitness-Werte der Generation mit der Funktion `calculate_fitness()` berechnet.
In `calculate_fitness()` wird mit Hilfe einer verschachtelten for-Schleife jede Kombination mit jeder anderen verglichen.
Für den Vergleich wird die Funktion `compare_two_chromosomes()` herangezogen. Dieser Funktion werden zwei Kombinationen übergeben. Dann wird überprüft, ob die zweite Kombination, die erste schlagen würde. Dafür werden die Distanzen jedes Hauses, die mit `distance_for_every_house()` berechnet werden, miteinander verglichen. Die Distanzen zwischen zwei Positionen auf dem Umfang des Kreises werden mit der Funktion `distance_between_two()` ermittelt.
Wenn die Fitness-Werte berechnet wurden, geht es weiter in der Funktion `ga()`. Mit der Funktion `get_best_chromosomes()` werden die Kombinationen ermittelt, die den geringsten Fitness-Wert der aktuellen Generation haben.
Nun muss die nächste Generation ermittelt werden. Dafür wird zunächst mit der Funktion `selection()` eine Auswahl von Kombinationen der vorherigen Generationen auf Basis des Fitness-Werts getroffen. Diese Auswahl wird dann mit den Funktionen `crossover()` und `mutation()` rekombiniert und mutiert. Die nächste Generation ist somit ermittelt.
Nun muss noch die Terminationsbedingung geprüft werden. Dafür wird die Funktion `termination()` ausgeführt. In dem Dictionary `overall_best_chromosomes` befinden sich die Kombinationen, die bereits

als beste Kombination in einer vorherigen Generation ermittelt wurde, mit der Häufigkeit, wie oft sie eine der besten Kombinationen war. Dieses Dictionary wird für die Prüfung der Terminationsbedingung verwendet. Wenn die Terminationsbedingung nicht erfüllt ist, ruft sich die Funktion `ga()` rekursiv auf. Nach Beendung des rekursiven Aufrufs muss noch das Ergebnis ermittelt werden. Dies übernimmt die Funktion `select_overall_winning_ones()`, in der mit der Funktion `most_occurences_of_chromosomes()` zunächst die Kombinationen ermittelt werden, die am häufigsten als beste Kombinationen einer Generation ausgezeichnet wurden. Dann wird das Verfahren wie in 1.4 beschrieben auf diese Kombinationen angewandt und ausgegeben, ob ein vermutlich stabiler Standort gefunden werden konnte, und wenn ja welcher, oder nicht.

3 Beispiele

Hier werden die Lösungen der Beispiele von der BwInf-Webseite vorgestellt.

3.1 eisbuden1.txt

```
Stabile Standorte sind vermutlich [[2, 5, 14], [2, 6, 14], [2, 7, 14], [2, 8, 14], [2, 9,
    14], [2, 10, 14], [2, 10, 15], [2, 11, 14], [2, 11, 15], [2, 12, 14], [2, 12, 15]]
```

3.2 eisbuden2.txt

```
Es konnten keine stabilen Standorte gefunden werden.
```

3.3 eisbuden3.txt

```
Stabile Standorte sind vermutlich [[0, 17, 42], [1, 17, 42]]
```

3.4 eisbuden4.txt

```
Es konnten keine stabilen Standorte gefunden werden.
```

3.5 eisbuden5.txt

```
Stabile Standorte sind vermutlich [[86, 124, 234], [84, 124, 233], [87, 126, 241], [85,
    126, 234], [83, 128, 231]]
```

3.6 eisbuden6.txt

```
Es konnten keine stabilen Standorte gefunden werden.
```

3.7 eisbuden7.txt

```
Stabile Standorte sind vermutlich [[126, 291, 418]]
```

4 Quellcode

```python
# Funktion zur Initialisierung der ersten Generation
def initial_population():
    population = []
    for _ in range(pop_size):
        chromosome = []
        # bis es drei Standorte in der aktuellen Kombination gibt
        while len(chromosome) < 3:
            # zufaellige Position wird ermittelt
            random_position = random.randint(0, umfang-1)
            # wenn der Standort noch nicht in der Kombination enthalten ist -> hinzufuegen
            # es soll nicht mehrere Eisbuden an einem Standort geben
            if random_position not in chromosome:
                chromosome.append(random_position)
        chromosome.sort()
        population.append(chromosome)
    # erste Generation zurueckgeben
    return population


# gibt Distanz zwischen zwei Positionen auf dem Umfang des Kreises zurueck
def distance_between_two(pos1, pos2):
    return min([umfang - max([pos1, pos2]) + min([pos1, pos2]), max([pos1, pos2]) - min([pos1, pos2])])


# gibt die minimale Distanz aller Haeuser zu einer Kombination von Eisbudenstandorten zurueck
def distance_for_every_house(chromosome):
    if str(chromosome) not in saved_distances:
        distances = []
        for h in haeuser:
            # fuer jeden Eisbudenstandort wird die Distanz berechnet, niedrigste wird ausgewaehlt
            dist1 = distance_between_two(h, chromosome[0])
            dist2 = distance_between_two(h, chromosome[1])
            dist3 = distance_between_two(h, chromosome[2])
            distances.append(min([dist1, dist2, dist3]))
        # berechnete Distanzen speichern, sodass sie nicht immmer wieder neue berechnet werden muessen
        saved_distances[str(chromosome)] = distances.copy()
        return distances
    else:
        return saved_distances[str(chromosome)]


# vergleicht zwei Kombination und gibt zurueck, ob erste Kombination in einer Abstimmung verlieren wuerde oder
nicht
def compare_two_chromosomes(c1, c2):
    distances_c1 = distance_for_every_house(c1)
    distances_c2 = distance_for_every_house(c2)

    # die Anzahl der Ja-Stimmen fuer eine Umverlegung wird nach den Regeln aus der Aufgabenstellung berechnet
    yes_votes = 0
    for i in range(len(distances_c1)):
        if distances_c1[i] > distances_c2[i]:
            yes_votes += 1

    # Wenn die Mehrheit Ja-Stimmen sind -> erste Komb. hat verloren
    if yes_votes > len(haeuser) / 2:
        return True
    return False


# Fitness-Funktion
def calculate_fitness(population):
    # setzt Grundfitness
    fitness = [len(population)-1]*len(population)
    # vergleicht jede Kombination mit allen anderen
    for i in range(len(population)):
        chromosome = population[i]
        for j in range(len(population)):
            if i != j:
                compare_chromosome = population[j]
                if compare_two_chromosomes(chromosome, compare_chromosome) or chromosome[0] ==
compare_chromosome[0]:
                    # verringert Fitness-Wert um 1, wenn die aktuelle Kombination nicht geschlagen wurde
                    fitness[j] -= 1
    # gibt Fitness aller Kombinationen zurueck
    return fitness
```

```python
# Fitness-Funktion
def calculate_fitness(population):
    # setzt Grundfitness
    fitness = [len(population)-1]*len(population)
    # vergleicht jede Kombination mit allen anderen
    for i in range(len(population)):
        chromosome = population[i]
        for j in range(len(population)):
            if i != j:
                compare_chromosome = population[j]
                if compare_two_chromosomes(chromosome, compare_chromosome) or chromosome[0] ==
compare_chromosome[0]:
                    # verringert Fitness-Wert um 1, wenn die aktuelle Kombination nicht geschlagen wurde
                    fitness[j] -= 1
    # gibt Fitness aller Kombinationen zurueck
    return fitness


# ermittelt die Kombinationen aus der aktuellen Generation, die den besten Fitness-Wert haben
def get_best_chromosomes(population, fitness):
    best_chromosomes = []
    for i in range(len(population)):
        # Wenn die Fitness der minimalen Fitness entspricht -> eine der besten Kombinationen der akt. Gen.
        if fitness[i] == min(fitness):
            dist = distance_for_every_house(population[i])
            if [population[i], dist] not in best_chromosomes:
                best_chromosomes.append([population[i], dist])
    # gibt "beste" Kombinationen zurueck
    return best_chromosomes


# Selektions-Funktion, erhält alte Generation und ermittelt den "Pool" der naechsten Generation
def selection(population, fitness):
    population_with_probabilites = []
    for i in range(len(population)):
        # Fuegt die Kombinationen gemaess der berechneten Anzahl auf grundlage des Fitness-Wertes hinzu
        population_with_probabilites += [copy.deepcopy(population[i])] * (pop_size-1 - fitness[i])
    mating_pool = []
    # waehlt zufaellig Kombinationen aus
    for _ in range(pop_size):
        choice = random.choice(population_with_probabilites)
        mating_pool.append(choice)
    return mating_pool


# Rekombinations-Funktion
def crossover(mating_pool):
    # iteriert ueber alle Kombination der aktuellen Generation
    for i in range(len(mating_pool)):
        # zufaellige Zahl um zu bestimmen, ob rekombiniert werden soll oder nicht
        random_number = random.randint(0, 99)/100
        if random_number < crossover_rate:
            # Partner wird zufaellig ausgewaehlt
            mating_partner = random.choice(mating_pool)
            if mating_partner != mating_pool[i]:
                # es wird zufaellig ausgewaehlt, welcher Teil behalten werden soll und welcher nicht
                splitting_index = random.choice([1, 2])
                mating_pool[i] = mating_pool[i][:splitting_index] + mating_partner[splitting_index:]
    return mating_pool


# Mutations-Funktion
def mutation(mating_pool):
    # es wird fuer jeden Standort entschieden, ob er mutieren soll oder nicht
    for i in range(len(mating_pool)):
        for j in range(len(mating_pool[i])):
            # zufaellige Zahl um zu bestimmen, ob mutiert werden soll oder nicht
            random_number = random.randint(0, 99)/100
            if random_number < mutation_rate:
                # setze neuen, zufaelligen Standort
                mating_pool[i][j] = random.randint(0, umfang-1)
        mating_pool[i].sort()
    return mating_pool
```

```python
# Terminationsbedingung
def termination(best_chromosomes, generation_counter):
    # ueber jede beste Kombination der aktuellen Generation wird iteriert
    for chromosome in best_chromosomes:
        # es wird gespeichert, dass diese Komb. zur besten Komb. gehoert
        if str(chromosome[0]) not in overall_best_chromosomes:
            overall_best_chromosomes[str(chromosome[0])] = 1
        else:
            overall_best_chromosomes[str(chromosome[0])] += 1
        if generation_counter > min_generation_count:
            # Wenn Terminationsbedingung eintrifft, wird beendet
            if overall_best_chromosomes[str(chromosome[0])] > generation_counter * termination_percentage:
                return True
    return False

# Main-Funktion, die sich rekursiv aufruft
def ga(population, generationen_counter):
    # Fitness-Werte werden berechnet
    fitness = calculate_fitness(population)
    # beste Kombinationen werden ermittelt
    best_chromosomes = sorted(get_best_chromosomes(population, fitness))
    # neuer "Pool" wird ausgewählt
    mating_pool = selection(population, fitness)
    # Rekombination wird angewandt
    mating_pool_after_crossover = crossover(copy.deepcopy(mating_pool))
    # Mutation wird angewandt
    mating_pool_after_mutation = mutation(copy.deepcopy(mating_pool_after_crossover))

    # Terminationsbedinung wird ueberprueft
    if not termination(best_chromosomes, generationen_counter):
        # Wiederaufruf der Funktion --> neue Generation
        ga(mating_pool_after_mutation, generationen_counter+1)
    else:
        return

# waehlt anhand der Kombinationen, die am haeufigsten als beste auftraten, das Ergebnis aus
def select_overall_winning_ones():
    # beste Kombinationen abrufen
    most_occurences = most_occurences_of_chromosomes()
    # Fitness-Werte berechnen
    fitness = calculate_fitness(most_occurences)
    # Wenn Fitness-Wert keiner Komb. 0 ist --> konnte kein stabiler Eisbudenstandort gefunden werden
    if min(fitness) > 0:
        print("Es konnte kein stabiler Standort gefunden werden")
    # gibt Komb. aus, die einen Fitness-Wert von 0 haben --> vermutlich stabile Standorte
    else:
        best_chromosomes = [x[0] for x in get_best_chromosomes(most_occurences, fitness)]
        print("Stabile Standorte sind vermutlich", best_chromosomes)


# sucht die Kombinationen, die am haeufigsten als "beste" Kombinationen vorkamen und gibt sie zurueck
def most_occurences_of_chromosomes():
    most_occurences = []
    for chromosome, occurences in overall_best_chromosomes.items():
        if len(most_occurences) < comparison_between_best:
            most_occurences.append([chromosome, occurences])
        else:
            for i in range(len(most_occurences)):
                if most_occurences[i][1] < occurences:
                    most_occurences[i] = [chromosome, occurences]
                    break

    for j in range(len(most_occurences)):
        most_occurences[j] = json.loads(most_occurences[j][0])
    return most_occurences


# Einlesen der Datei
path = "eisbuden1.txt"
file = open(path, "r")

first = [int(x) for x in file.readline().split(" ") if x != "\n"]
umfang = first[0]
anzahl_haeuser = first[1]
haeuser = [int(x) for x in file.readline().split(" ") if x != "\n"]
saved_distances = {}
overall_best_chromosomes = {}

# Variablen definiert
pop_size = 55
mutation_rate = 0.085
crossover_rate = 0.85
min_generation_count = 15
comparison_between_best = 20
termination_percentage = 0.06

# Aufrufen der Hauptfunktionen
ga(initial_population(), 0)
select_overall_winning_ones()
```