Algorithmen zur Lösung klassischer Schach- und Rätselprobleme Analyse existierender Algorithmen und Entwicklung neuer Lösungsansätze

Teodor Marinov, 201222003

Stefan Georgiev, 201222011

Deyvid Popov, 201221006

Technische Universität Sofia

Fakultät für deutsche Ingenieur- und Betriebswirtschaftsausbildung

Informatik

Logik

09.05.2023

Inhaltsverzeichnis

Einleitung

Im Bereich der Informatik und Mathematik haben klassische Schach- und Rätselprobleme schon immer Neugier geweckt und die Köpfe von Enthusiasten herausgefordert. Diese Kursarbeit hat zum Ziel, die Feinheiten von drei solch faszinierenden Problemen zu untersuchen: die Acht-Damen-Aufgabe, die Ritter-Tour und der Turm von Hanoi. Wir werden uns nicht nur in die bestehenden Algorithmen vertiefen, die versuchen, diese rätselhaften Herausforderungen zu bewältigen, sondern auch in unerforschtes Gebiet vordringen, indem wir unsere eigenen Algorithmen entwerfen. Mit dem Hauptziel, neuartige Lösungen für diese altbekannten Rätsel zu entwickeln, lädt diese Kursarbeit die Leser ein, sich auf eine fesselnde Reise durch die Welt der algorithmischen Problemlösung zu begeben.

Probleme und existierende Lösungen

Acht-Damen-Aufgabe

Die Acht-Damen-Aufgabe ist ein klassisches Schachrätsel, bei dem es darum geht, acht Damen auf einem 8x8-Schachbrett so zu platzieren, dass keine zwei Damen sich gegenseitig bedrohen. Das bedeutet, dass keine zwei Damen dieselbe Reihe, Spalte oder Diagonale teilen dürfen.

Existierende Lösungen:

- **Breitensuche (BFS):** Ein Ansatz zur Lösung des Achtdamenproblems, der alle möglichen Damenaufstellungen Ebene für Ebene untersucht und den Suchbaum in der Breite erweitert.

```python
from collections import deque

def bfs_solve():
    # Initialisiere die Warteschlange mit einem leeren Schachbrett und der ersten Spalte
    queue = deque([([], 0)])

    while queue:
        # Entferne das erste Element aus der Warteschlange
        queens, column = queue.popleft()

        # Wenn alle 8 Königinnen platziert sind, ist das Problem gelöst
        if column == 8:
            return queens

        # Gehe alle Zeilen der aktuellen Spalte durch
        for row in range(8):
            # Prüfe, ob die Platzierung der Königin gültig ist
            if is_valid(queens, row, column):
                # Erstelle eine Kopie der aktuellen Königinnen und füge die neue Königin hinzu
                new_queens = queens.copy()
                new_queens.append((row, column))
                # Füge die neue Platzierung der Königinnen und die nächste Spalte der Warteschlange hinzu
                queue.append((new_queens, column + 1))

    return None

def is_valid(queens, row, column):
    # Gehe alle platzierten Königinnen durch
    for q_row, q_col in queens:
        # Prüfe, ob die neue Königin von einer anderen Königin angegriffen wird
        if q_row == row or q_col == column or abs(row - q_row) == abs(column - q_col):
            return False
    return True

solution = bfs_solve()
for row in range(8):
    board_row = ['.'] * 8
```

```python
    for queen in solution:
        if queen[0] == row:
            board_row[queen[1]] = 'Q'
    print("".join(board_row))
```

- **Tiefensuche (DFS):** DFS ist eine übliche Methode zur Lösung der Acht-Damen-Aufgabe. Dabei werden die Damen nacheinander auf dem Schachbrett platziert und bei Konflikten zurückgesetzt.

```python
def is_safe(board, row, col):
    # Prüfe auf horizontale Angriffe
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Prüfe auf diagonale Angriffe oben links
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Prüfe auf diagonale Angriffe unten links
    for i, j in zip(range(row, len(board), 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True

def solve_eight_queens_dfs(board, col):
    # Wenn alle Königinnen platziert sind, ist das Problem gelöst
    if col >= len(board):
        return True

    # Gehe durch alle Zeilen der aktuellen Spalte
    for i in range(len(board)):
        if is_safe(board, i, col):
            board[i][col] = 1
            if solve_eight_queens_dfs(board, col + 1):
                return True
            board[i][col] = 0

    return False

def print_board(board):
    for row in board:
        print(" ".join(str(cell) for cell in row))

board = [[0 for _ in range(8)] for _ in range(8)]

if solve_eight_queens_dfs(board, 0):
    print("Lösung gefunden:")
    print_board(board)
```

```python
else:
    print("Keine Lösung gefunden")
```

- **Backtracking mit Vorwärtsprüfung:** Dieser Ansatz ist eine Erweiterung der DFS-Methode, bei dem der Algorithmus nicht nur zurückverfolgt, sondern auch vor dem Platzieren einer Dame auf Konflikte prüft und so den Suchraum reduziert.

```python
# Initialisiere das Schachbrett
chess_board = [[0]*8 for _ in range(8)]

# Prüft, ob eine Königin in den angegebenen Koordinaten (i, j) angegriffen wird
def attack(i, j):
    # Prüft horizontale und vertikale Angriffe
    for k in range(0,8):
        if chess_board[i][k]==1 or chess_board[k][j]==1:
            return True
    # Prüft diagonale Angriffe
    for k in range(0,8):
        for l in range(0,8):
            if (k+l==i+j) or (k-l==i-j):
                if chess_board[k][l]==1:
                    return True
    return False

# Löst das Eight-Queens-Problem mit Backtracking und Forward Checking
def solve(n):
    # Wenn alle Königinnen platziert sind, ist das Problem gelöst
    if n==0:
        return True
    # Gehe durch alle Zellen des Schachbretts
    for i in range(0,8):
        for j in range(0,8):
            # Wenn die Zelle nicht angegriffen wird und keine Königin enthält
            if (not(attack(i,j))) and (chess_board[i][j]!=1):
                # Platziere eine Königin in der Zelle
                chess_board[i][j] = 1
                # Rufe rekursiv die Funktion auf, um die verbleibenden Königinnen zu platzieren
                if solve(n-1)==True:
                    return True
                # Wenn die Platzierung fehlschlägt, entferne die Königin aus der Zelle
                chess_board[i][j] = 0
    return False

# Löse das Eight-Queens-Problem
solve(8)

# Drucke das resultierende Schachbrett
for i in chess_board:
    print(i)
```

Springerproblem

Das Springerproblem besteht darin, einen Springer auf einem leeren Schachbrett so zu bewegen, dass er jedes Feld genau einmal besucht. Die Herausforderung besteht darin, einen geschlossenen Weg zu finden, bei dem der Springer am Ende auf einem Feld landet, das einen legalen Zug vom Startpunkt entfernt ist.

Existierende Lösungen:

- **Tiefensuche (DFS):** DFS erforscht Züge, indem es tiefer in den Suchbaum eintaucht, bevor es zurückverfolgt wird. Es ist im Allgemeinen schneller, findet aber möglicherweise nicht in allen Fällen eine Lösung.

```python
import random

# Schachbrett ausgeben
def print_board():
    for row in chess_board:
        for cell in row:
            print(f"{cell:2}", end=" ")
        print()

# Mögliche nächste Züge für den Springer erhalten
def get_possibilities(x, y):
    pos_x = (2, 1, 2, 1, -2, -1, -2, -1)
    pos_y = (1, 2, -1, -2, 1, 2, -1, -2)
    possibilities = []

    for i in range(8):
        next_x, next_y = x + pos_x[i], y + pos_y[i]
        if 0 <= next_x < 8 and 0 <= next_y < 8 and chess_board[next_x][next_y] == 0:
            possibilities.append((next_x, next_y))

    return possibilities

# Löse das Rittersprung-Problem mit DFS und Warnsdorffs Regel
def knight_tour_dfs(x, y, move_count):
    if move_count > 64:
        return True

    next_moves = get_possibilities(x, y)
    next_moves.sort(key=lambda move: len(get_possibilities(move[0], move[1])))  # Warnsdorff rule

    for next_x, next_y in next_moves:
        chess_board[next_x][next_y] = move_count
        if knight_tour_dfs(next_x, next_y, move_count + 1):
            return True
        chess_board[next_x][next_y] = 0  # Backtrack

    return False

# Schachbrett zurücksetzen
chess_board = [[0 for _ in range(8)] for _ in range(8)]

# Zufällige Startposition auswählen
start_x, start_y = random.randint(0, 7), random.randint(0, 7)
chess_board[start_x][start_y] = 1

# Löse das Rittersprung-Problem mit DFS und Warnsdorffs Regel
if knight_tour_dfs(start_x, start_y, 2):
    print("Solution found with random starting position:")
    print_board()
else:
    print("No solution found")
```

- **Warnsdorffs Heuristik:** Dies ist eine heuristikbasierte Methode, bei der der nächste Zug mit der geringsten Anzahl von Folgezügen ausgewählt wird. Es ist schneller und effizienter als die anderen Methoden, findet jedoch nicht immer eine Lösung.

```python
# Schachbrett initialisieren
def create_board(start_x, start_y):
    chess_board = [[0 for _ in range(8)] for _ in range(8)]
    chess_board[start_x][start_y] = 1
    return chess_board

start_x, start_y = 0, 0
chess_board = create_board(start_x, start_y)

# Funktion zum Ausgeben des Schachbretts
def print_board():
    for row in chess_board:
        for cell in row:
            print(f"{cell:2}", end=" ")
        print()

# Funktion zum Ermitteln der möglichen Züge für den Springer in der Position (x, y)
def get_possibilities(x, y):
    pos_x = (2, 1, 2, 1, -2, -1, -2, -1)
    pos_y = (1, 2, -1,-2, 1, 2, -1, -2)
    possibilities = []
    for i in range(8):
        if x+pos_x[i] ≥ 0 and x+pos_x[i] ≤ 7 and y+pos_y[i] ≥ 0 and y+pos_y[i] ≤ 7 and chess_board[x+pos_x[i]][y+pos_y[i]] == 0:
            possibilities.append([x+pos_x[i], y+pos_y[i]])

    return possibilities

# Funktion zum Lösen des Rittertourenproblems
def solve():
    counter = 2
    x = 0
    y = 0
    for _ in range(63):
        pos = get_possibilities(x, y)
        minimum = pos[0]
        for p in pos:
            if len(get_possibilities(p[0], p[1])) ≤ len(get_possibilities(minimum[0], minimum[1])):
                minimum = p
        x = minimum[0]
        y = minimum[1]
        chess_board[x][y] = counter
        counter += 1

# Löse das Rittertourenproblem und gebe das resultierende Schachbrett aus
solve()
print_board()
```

Türme von Hanoi

Die Türme von Hanoi sind ein klassisches Rätsel, bei dem es darum geht, einen Stapel von Scheiben von einem Pflock auf einen anderen zu bewegen, wobei ein dritter Pflock als Zwischenstation dient, und bestimmte Regeln befolgt werden müssen: Es darf nur eine Scheibe auf einmal bewegt werden, und eine größere Scheibe darf nicht auf einer kleineren liegen.

Existierende Lösung:

- **Rekursiver Algorithmus:** Der häufigste Ansatz zur Lösung der Türme von Hanoi besteht darin, einen rekursiven Algorithmus zu verwenden, der das Problem in kleinere Teilprobleme zerlegt. Diese Methode garantiert eine Lösung, kann jedoch bei einer größeren Anzahl von Scheiben langsam sein.

```python
def tower_of_hanoi(n, source, destination, auxiliary):
    if n == 1:
        # Bewegung der Scheibe 1 von der Quelle zum Ziel ausgeben
        print(f"Bewege Scheibe 1 von Quelle {source} zum Ziel {destination}")
        return

    # Bewege die obersten n-1 Scheiben von der Quelle zum Hilfsstift mit dem Zielstift
    tower_of_hanoi(n-1, source, auxiliary, destination)
```

```python

    # Bewege die n-te Scheibe von der Quelle zum Zielstift
    print(f"Bewege Scheibe {n} von Quelle {source} zum Ziel {destination}")

    # Bewege die n-1 Scheiben vom Hilfsstift zum Zielstift mit dem Quellstift
    tower_of_hanoi(n-1, auxiliary, destination, source)

n = 4
tower_of_hanoi(n, 'A', 'B', 'C')
```

Unsere Lösungen

RegalRunner

Der RegalRuler-Algorithmus ist ein einzigartiger und innovativer Ansatz zur Lösung des 8-Königinnen-Problems. Er kombiniert die Konzepte der genetischen Algorithmen mit einer benutzerdefinierten Fitnessfunktion und bietet so eine effiziente und effektive Lösung. Der Algorithmus trägt den Namen "RegalRuler", da es darum geht, Königinnen so auf dem Brett zu platzieren, dass keine Königin eine andere bedroht, was die königliche Natur der Königinnen symbolisiert.

Der im RegalRuler verwendete genetische Algorithmus ahmt den Prozess der natürlichen Selektion nach, bei dem die am besten geeigneten Individuen für Reproduktion, Crossover und Mutation ausgewählt werden, um die nächste Generation zu erzeugen. Die Fitnessfunktion bewertet eine Lösung anhand der Anzahl der Konflikte zwischen den Königinnen, wobei das Ziel darin besteht, die Konflikte zu minimieren. Die Kombination dieser Techniken bietet eine leistungsfähige und einzigartige Methode zur Lösung des Problems.

Der RegalRuler-Algorithmus hat eine durchschnittliche Zeitkomplexität von O(n^2), wobei n die Anzahl der Königinnen ist. Diese Komplexität kann jedoch je nach den für den genetischen Algorithmus verwendeten Parametern, wie z. B. Populationsgröße und Mutationsrate, variieren.

```python
import random

# Fitnessfunktion zur Bewertung einer Lösung
def fitness(solution):
    conflicts = 0
    n = len(solution)

    for i in range(n):
        for j in range(i + 1, n):
            if solution[i] == solution[j]:
                conflicts += 1
            elif abs(solution[i] - solution[j]) == abs(i - j):
                conflicts += 1

    return conflicts

# Crossover-Operation für den genetischen Algorithmus
def crossover(parent1, parent2):
    crossover_point = random.randint(1, len(parent1) - 1)
    return parent1[:crossover_point] + parent2[crossover_point:]

# Mutationsoperation für den genetischen Algorithmus
def mutate(solution):
    index = random.randint(0, len(solution) - 1)
    value = random.randint(1, len(solution))
    mutated_solution = solution[:]
    mutated_solution[index] = value
    return mutated_solution

```

```python
# RegalRuler-Funktion, die einen genetischen Algorithmus verwendet, um das 8-Königinnen-Problem zu lösen
def regal_ruler(population_size, max_generations, mutation_rate):
    n = 8

    # Erzeugen Sie die anfängliche Bevölkerung
    population = [random.sample(range(1, n + 1), n) for _ in range(population_size)]

    for _ in range(max_generations):
        population.sort(key=fitness)

        # Überprüfen Sie, ob eine Lösung gefunden wurde
        if fitness(population[0]) == 0:
            return population[0]

        # Wählen Sie die besten Individuen für Crossover aus
        selected = population[:population_size // 2]

        # Crossover und Mutation durchführen
        new_population = []
        for _ in range(population_size):
            parent1 = random.choice(selected)
            parent2 = random.choice(selected)
            offspring = crossover(parent1, parent2)
            if random.random() < mutation_rate:
                offspring = mutate(offspring)
            new_population.append(offspring)

        population = new_population

    return None

# Lösen Sie das 8-Königinnen-Problem mit dem RegalRuler-Algorithmus
solution = regal_ruler(100, 1000, 0.1)
if solution:
    print("Solution found:", solution)
else:
    print("No solution found")
```

ShadowWalker

ShadowWalker ist ein origineller Algorithmus zur Lösung des Rittersprungproblems. Der Name ShadowWalker repräsentiert die heimliche und strategische Natur eines Ritters, der sich auf dem Schachbrett bewegt. Der Algorithmus verwendet einen Tiefensuche-Ansatz in Kombination mit einer heuristischen Priorisierung der Abstand zum Zentrum. Die Priorisierung der Abstand zum Zentrum führt dazu, dass der Ritter zuerst das Zentrum des Spielfelds erkundet, wodurch die Wahrscheinlichkeit, eine Lösung zu finden, erhöht wird. Dies liegt daran, dass die zentralen Felder dem Ritter mehr Zugmöglichkeiten bieten und so zu flexibleren Pfaden führen. Die Zeitkomplexität des ShadowWalker-Algorithmus beträgt O(8^N), wobei N die Anzahl der Felder auf dem Schachbrett ist, und die Raumkomplexität beträgt O(N).

```python
import random

# Drucke das Schachbrett (Print the chessboard)
def print_board(board):
    for row in board:
        for cell in row:
            print(f"{cell:2}", end=" ")
        print()

# Hole die möglichen Züge
def get_possibilities(x, y, board):
    pos_x = (2, 1, 2, 1, -2, -1, -2, -1)
    pos_y = (1, 2, -1, -2, 1, 2, -1, -2)
    possibilities = []

    for i in range(8):
        next_x, next_y = x + pos_x[i], y + pos_y[i]
        if 0 ≤ next_x < 8 and 0 ≤ next_y < 8 and board[next_x][next_y] == 0:
            possibilities.append((next_x, next_y))
```

```python

    return possibilities

# Berechne die Distanz zum Zentrum
def distance_to_center(x, y):
    return abs(3.5 - x) + abs(3.5 - y)

# Kombiniere Warnsdorffs Regel und Zentrumsdistanzpriorisierung
def combined_heuristic(x, y, move_count, board):
    if move_count > 64:
        return True

    next_moves = get_possibilities(x, y, board)
    next_moves.sort(key=lambda move: (len(get_possibilities(move[0], move[1], board)), distance_to_center(move[0], move[1])))

    for next_x, next_y in next_moves:
        board[next_x][next_y] = move_count
        if combined_heuristic(next_x, next_y, move_count + 1, board):
            return True
        board[next_x][next_y] = 0  # Zurückverfolgen

    return False

# Schachbrett zurücksetzen
chess_board = [[0 for _ in range(8)] for _ in range(8)]

# Zufällige Startposition wählen
start_x, start_y = random.randint(0, 7), random.randint(0, 7)
chess_board[start_x][start_y] = 1

# Löse das Springertour-Problem mit dem ShadowWalker-Algorithmus
if combined_heuristic(start_x, start_y, 2, chess_board):
    print("Lösung gefunden mit zufälliger Startposition")
    print_board(chess_board)
else:
    print("Keine Lösung gefunden")
```

DarkAbyss

DarkAbyss ist ein einzigartiger und origineller Algorithmus zur Lösung des Turms von Hanoi Problems. Der Name DarkAbyss repräsentiert die mysteriöse und tiefe Natur des Problems. Der Algorithmus verwendet einen iterativen Ansatz in Kombination mit Bitoperationen, um den Ausgangs-, Hilfs- und Zielstab für jeden Zug zu bestimmen. Dies macht es zu einer effizienten Lösung mit einer Zeitkomplexität von O(2^n) und einer Raumkomplexität von O(1).

```python
def dark_abyss(n):
    # Berechne die Gesamtzahl der Züge
    num_moves = 2 ** n - 1
    pegs = ["A", "B", "C"]

    # Durchlaufe jeden Zug
    for move in range(1, num_moves + 1):
        # Bestimme den Ausgangsstab mit Hilfe von Bitoperationen
        from_peg = pegs[(move & move - 1) % 3)]
        # Bestimme den Zielstab mit Hilfe von Bitoperationen
        to_peg = pegs[(((move | move - 1) + 1) % 3)]
        # Drucke den Zug
        print(f"Move disk from peg {from_peg} to peg {to_peg}")

n = 4
dark_abyss(n)
```

Schluss

Zusätzliche Ressourcen

Um auf den Quellcode und die Dateien für dieses Projekt zuzugreifen, besuchen Sie bitte unser Github-Repository unter folgendem Link:

https://github.com/DeyvidTheWise/Logic-Course-Assignment

Zusammenfassung der Ergebnisse

In diesem Projekt haben wir verschiedene Algorithmen und Methoden untersucht, um unterschiedliche Probleme wie das Ritterproblem, den Turm von Hanoi und das 8-Königinnen-Problem zu lösen. Es ist eine anspruchsvolle Aufgabe, völlig neue Algorithmen zu erstellen, da bereits viele Lösungen existieren. Durch die Kombination verschiedener Regeln und Algorithmen können wir jedoch innovative Ansätze entwickeln, die möglicherweise die Effizienz und Effektivität bei der Lösung dieser Probleme verbessern können.

Unsere Lösungen, die Algorithmen ShadowWalker, DarkAbyss und RegalRuler, sind Beispiele für diesen innovativen Ansatz. Durch die Kombination vorhandener Methoden und die Einführung neuer Techniken haben wir einzigartige und effiziente Algorithmen entwickelt, die sich diesen klassischen Problemen stellen. Diese Algorithmen demonstrieren die Kraft der Kombination mehrerer Techniken und des kreativen Denkens bei der Lösung komplexer Probleme.

Ausblick und mögliche Erweiterungen

In der Zukunft besteht Potenzial für weitere Verbesserungen und Optimierungen dieser Algorithmen. Darüber hinaus können neue Techniken und Ansätze erforscht werden, um noch effizientere und effektivere Lösungen zu entwickeln. Durch kontinuierliche Überprüfung und Anpassung unserer Methoden können wir zur fortlaufenden Entwicklung und Weiterentwicklung der Informatik und der Problemlösungstechniken beitragen.