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
    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)
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 i in range(8):
        for j in range(8):
            print(chess_board[i][j], end=" ")
        print("\n")
```

```python
# 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()
```