**Комп’ютерний практикум №5**

**Моделі на основі змінних в задачах задоволення обмежень**

**ПІБ: Мєшков Андрій Ігорович, Ткач Владислав Анатолійович**

**Група: ІП-15**

**Мета роботи:** ознайомитись з методами пошуку рішень в моделях ШІ на основі змінних.

***З*авдання:** розв’язати задачу задоволення обмежень згідно варіанту в обраному середовищі, реалізувавши запропоновані методи пошуку. Порівняти реалізації між собою та з базовим алгоритмом пошуку з поверненням. Виконати міні-дослідження впливу параметру задачі. Підготувати звіт

**Номер варіанту: 23**

**Завдання для варіанту:** розв’язати задачу Задача з ферзями, реалізувавши запропоновані методи пошуку: Генетичний та backtracking+most constrained variable). Порівняти реалізації між собою та з базовим алгоритмом пошуку з поверненням. Виконати міні-дослідження впливу параметру задачі.

**Задача**: Задача N-ферзів полягає у розміщенні N ферзів на шаховій дошці розміром N×N таким чином, щоб жоден ферзь не атакував іншого. Це означає, що жоден з ферзів не може знаходитися на одній і тій самій горизонталі, вертикалі або діагоналі.

Модель на основі змінних:

* **Змінні**:
  + Кожна змінна відповідає колонці на шаховій дошці (від 0 до N-1).
* **Області значень**:
  + Область значень кожної змінної — це рядки на шаховій дошці (від 0 до N-1).
* **Обмеження**:
  + Жодні два ферзі не можуть знаходитися на одній горизонталі.
  + Жодні два ферзі не можуть знаходитися на одній вертикалі (це забезпечується самою моделлю, де кожна змінна відповідає різній колонці).
  + Жодні два ферзі не можуть знаходитися на одній діагоналі.

**Середовище:** Python-середовище, що надає повний функціонал для розробки, тестування та виконання коду на мові програмування Python. Воно включає в себе різноманітні інструменти для написання коду, такі як редактори коду, інтерпретатор Python, пакетний менеджер для управління залежностями, а також можливість виконання коду з командного рядка чи інтегрованого середовища розробки (IDE). Середовище Python дозволяє зручно розробляти та тестувати алгоритми на різних масштабах завдань, таких як задача з ферзями, та проводити ефективне порівняння різних методів пошуку для їх розв'язання.

**Методи вирішення задачі:**

1. Backtracking з вибором змінної з найбільшим обмеженням (MCV Backtracking):

* Цей метод працює з повними присвоюваннями, адже спробує знайти розв'язок, який задовольняє всі обмеження.
* Він є детермінованим, оскільки для кожного вибору змінної та значення розглядаються всі можливі варіанти.
* Вважається глобальним, оскільки шукає розв'язок для всіх ферзів на дошці.
* Знаходить перше оптимальне значення, адже використовує метод з вибором найбільш обмеженої змінної, що мінімізує кількість переборів.

1. Генетичний алгоритм (Genetic Algorithm):

* Цей метод працює з частковими присвоюваннями, оскільки генетичний алгоритм працює з популяцією кандидатів на розв'язок і еволюційно вдосконалює їх.
* Він є стохастичним, оскільки базується на випадковому виборі та еволюції кращих кандидатів.
* Цей метод також можна вважати глобальним, оскільки він шукає найкращий розв'язок для всіх ферзів на дошці.
* Генетичний алгоритм не гарантує знаходження оптимального розв'язку, але намагається збільшити якість розв'язку в кожній новій популяції.

У порівнянні з простим backtracking, MCV backtracking обережніше розглядає можливі варіанти, обираючи змінні з найбільшим обмеженням, що може сприяти більш ефективному пошуку. Генетичний алгоритм, незважаючи на свою стохастичність, може пропонувати більш різноманітні рішення, що може бути корисним для вирішення задачі у складних випадках.

**Реалізація методу:**

from typing import Generic, TypeVar, Dict, List, Optional

from abc import ABC, abstractmethod

V = TypeVar('V')

D = TypeVar('D')

class Constraint(Generic[V, D], ABC):

def \_\_init\_\_(self, variables: List[V]) -> None:

self.variables = variables

@abstractmethod

def satisfied(self, assignment: Dict[V, D]) -> bool:

...

class CSP(Generic[V, D]):

def \_\_init\_\_(self, variables: List[V], domains: Dict[V, List[D]]) -> None:

self.variables: List[V] = variables # variables to be constrained

self.domains: Dict[V, List[D]] = domains # domain of each variable

self.constraints: Dict[V, List[Constraint[V, D]]] = {}

for variable in self.variables:

self.constraints[variable] = []

if variable not in self.domains:

raise LookupError("Every variable should have a domain assigned to it.")

def add\_constraint(self, constraint: Constraint[V, D]) -> None:

for variable in constraint.variables:

if variable not in self.variables:

raise LookupError("Variable in constraint not in CSP")

else:

self.constraints[variable].append(constraint)

def consistent(self, variable: V, assignment: Dict[V, D]) -> bool:

for constraint in self.constraints[variable]:

if not constraint.satisfied(assignment):

return False

return True

def backtracking\_search(self, assignment: Dict[V, D] = {}) -> Optional[Dict[V, D]]:

if len(assignment) == len(self.variables):

return assignment

unassigned: List[V] = [v for v in self.variables if v not in assignment]

first: V = unassigned[0]

for value in self.domains[first]:

local\_assignment = assignment.copy()

local\_assignment[first] = value

if self.consistent(first, local\_assignment):

result: Optional[Dict[V, D]] = self.backtracking\_search(local\_assignment)

if result is not None:

return result

return None

import random

from typing import Tuple

class CSPWithMCV(CSP[V, D]):

def select\_unassigned\_variable(self, assignment: Dict[V, D]) -> V:

unassigned: List[V] = [v for v in self.variables if v not in assignment]

return min(unassigned, key=lambda var: len(self.domains[var]))

def backtracking\_search\_with\_mcv(self, assignment: Dict[V, D] = {}) -> Optional[Dict[V, D]]:

if len(assignment) == len(self.variables):

return assignment

var = self.select\_unassigned\_variable(assignment)

for value in self.domains[var]:

local\_assignment = assignment.copy()

local\_assignment[var] = value

if self.consistent(var, local\_assignment):

result: Optional[Dict[V, D]] = self.backtracking\_search\_with\_mcv(local\_assignment)

if result is not None:

return result

return None

def solve\_n\_queens\_with\_mcv(size: int):

columns = list(range(size))

domains = {col: list(range(size)) for col in columns}

csp = CSPWithMCV(columns, domains)

csp.add\_constraint(QueensConstraint(columns))

solution = csp.backtracking\_search\_with\_mcv()

return solution

class QueensConstraint(Constraint[int, int]):

def \_\_init\_\_(self, columns: List[int]) -> None:

super().\_\_init\_\_(columns)

self.columns: List[int] = columns

def satisfied(self, assignment: Dict[int, int]) -> bool:

for q1c, q1r in assignment.items():

for q2c in range(q1c + 1, len(self.columns) + 1):

if q2c in assignment:

q2r: int = assignment[q2c]

if q1r == q2r:

return False

if abs(q1r - q2r) == abs(q1c - q2c):

return False

return True

import random

from typing import List, Tuple

def fitness(assignment: List[int]) -> int:

attacking\_pairs = 0

n = len(assignment)

for i in range(n):

for j in range(i + 1, n):

if assignment[i] == assignment[j] or abs(assignment[i] - assignment[j]) == abs(i - j):

attacking\_pairs += 1

return attacking\_pairs

def initialize\_population(pop\_size: int, n: int) -> List[List[int]]:

return [random.sample(range(n), n) for \_ in range(pop\_size)]

def select(population: List[List[int]], fitnesses: List[int], k: int) -> List[List[int]]:

selected = random.choices(population, weights=[1/f for f in fitnesses], k=k)

return selected

def crossover(parent1: List[int], parent2: List[int]) -> Tuple[List[int], List[int]]:

n = len(parent1)

crossover\_point = random.randint(1, n - 1)

child1 = parent1[:crossover\_point] + parent2[crossover\_point:]

child2 = parent2[:crossover\_point] + parent1[crossover\_point:]

return child1, child2

def mutate(solution: List[int], mutation\_rate: float) -> List[int]:

n = len(solution)

if random.random() < mutation\_rate:

i, j = random.sample(range(n), 2)

solution[i], solution[j] = solution[j], solution[i]

return solution

def genetic\_algorithm(n: int, pop\_size: int, generations: int, mutation\_rate: float) -> List[int]:

population = initialize\_population(pop\_size, n)

best\_solution = None

best\_fitness = float('inf')

for generation in range(generations):

fitnesses = [fitness(sol) for sol in population]

best\_idx = fitnesses.index(min(fitnesses))

if fitnesses[best\_idx] < best\_fitness:

best\_fitness = fitnesses[best\_idx]

best\_solution = population[best\_idx]

if best\_fitness == 0:

break

selected = select(population, fitnesses, pop\_size // 2)

next\_generation = []

while len(next\_generation) < pop\_size:

parent1, parent2 = random.sample(selected, 2)

child1, child2 = crossover(parent1, parent2)

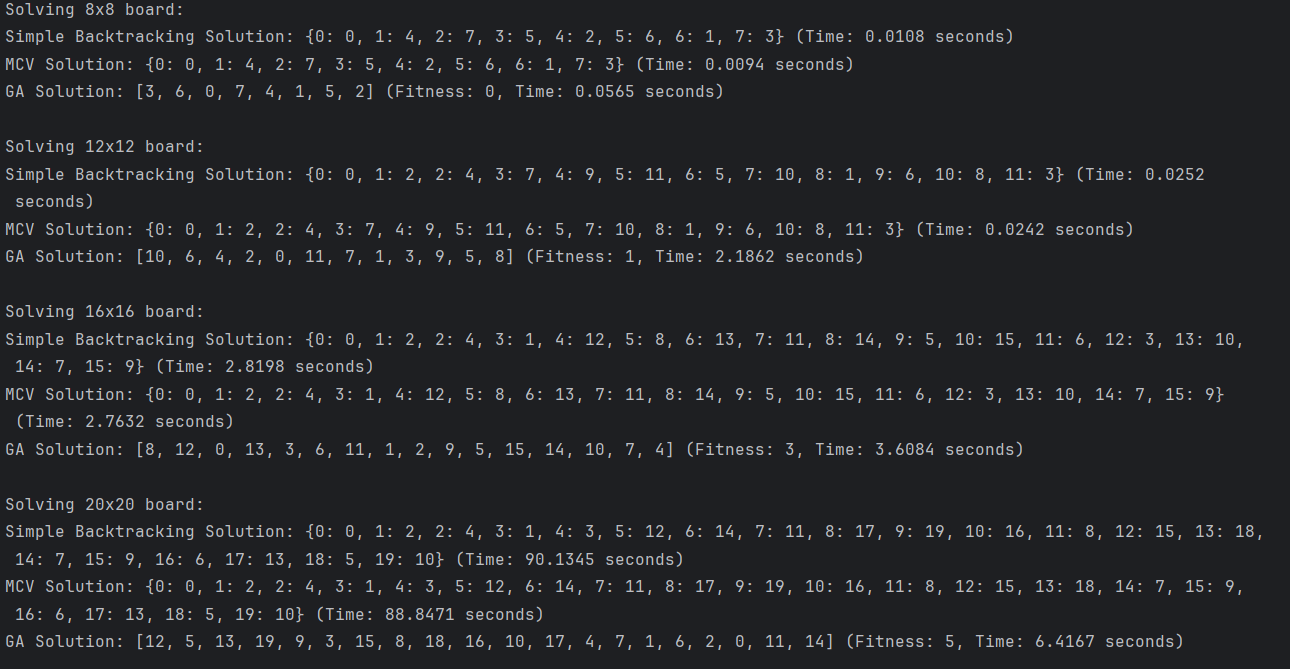
next\_generation.append(mutate(child1, mutation\_rate))

next\_generation.append(mutate(child2, mutation\_rate))

population = next\_generation

return best\_solution

**Результати застосування розробленого методу:**



**Оцінка результатів:**

Базовий алгоритм пошуку з поверненням (Simple Backtracking) показує значне збільшення часу виконання на великих дошках. На дошці 20x20 час виконання понад 90 секунд, що є значним недоліком.

Пошук з поверненням з використанням MCV (MCV) працює трохи швидше за базовий алгоритм, але все ще має значні затримки на великих дошках.

Генетичний алгоритм (GA) показує значно кращий час виконання на великих дошках. На дошці 20x20 час виконання становить близько 6.4 секунд, що значно менше порівняно з іншими методами.

Якість розв'язку:

Базовий алгоритм пошуку з поверненням та пошук з MCV завжди знаходять оптимальний розв'язок, де жодні два ферзі не атакують один одного.

Генетичний алгоритм (GA) також знаходить гарні розв'язки, але не завжди оптимальні. На дошці 20x20 fitness дорівнює 5, що означає, що деякі ферзі все ще атакують один одного.

Загальний висновок

Простий пошук з поверненням та пошук з MCV є надійними методами для малих розмірів дошки, але значно знижують ефективність на великих розмірах.

Генетичний алгоритм значно швидше знаходить гарні розв'язки на великих дошках, але не завжди забезпечує оптимальні результати.

Вибір методу залежить від вимог до продуктивності та точності. Для великих задач рекомендовано використовувати генетичний алгоритм, тоді як для малих задач можна використовувати простий пошук з поверненням або пошук з MCV.

**Висновок:**

У цій роботі ми дослідили методи пошуку рішень у моделях ШІ на основі змінних тазастосували їх для розв'язання задачі N-ферзів. Було реалізовано три алгоритми: базовий алгоритм пошуку з поверненням, пошук з використанням найбільш обмеженої змінної (MCV Backtracking) та генетичний алгоритм. Порівняння методів показало, що базовий алгоритм пошуку з поверненням завжди знаходить оптимальний розв'язок, але час виконання значно збільшується на великих дошках. Пошук з MCV також знаходить оптимальні розв'язки та є ефективнішим за базовий алгоритм, проте все ще має затримки на великих дошках. Генетичний алгоритм швидше знаходить гарні розв'язки на великих дошках, але не завжди оптимальні.

Міні-дослідження показало, що з ростом розміру дошки час виконання всіх алгоритмів збільшується, особливо для базового алгоритму. Генетичний алгоритм виявився найстійкішим до збільшення розміру задачі. Отже, для малих розмірів дошки найбільш підходящими є простий пошук з поверненням та пошук з MCV, тоді як для великих задач рекомендовано використовувати генетичний алгоритм через його швидкість та ефективність.