МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КУБГУ»)**

**Факультет компьютерных технологий и прикладной математики**

**Кафедра вычислительных технологий**

**Отчет**

**по индивидуальному заданию №2**

**««Максимальное независимое множество. Задача оптимизации»**

**по курсу**

**«ОЦЕНКА СЛОЖНОСТИ АЛГОРИТМОВ»**

Работу выполнил

Студент 36 группы

Корнилов К.А.

Преподаватель:

Лапина О.Н.

**Доказательство принадлежности классу NP**

Пусть известен граф G(V,E), |V|=m. На первом этапе генерируется множество V1 из  g<=m вершин и производится проверка того, является ли это множество независимым множеством вершин графа. На втором этапе производятся перебор всех вершин множества V/V1. Таких вершин всего m-g. На каждой итерации перебора одна из этих вершин добавляется в V1. Далее производится проверка того, является ли новое множество независимым множеством вершин графа. Если ни одно из таких множеств не является независимым множеством графа, то множество V1 будет являться максимальным независимым множеством графа. Построим формальный алгоритм генерации множества V1:

set\_g=new Array(g); a=new Set(); b=new Set(); nu=new Array(m);

for i=1..m{

nu[i]=i;

a.add(i);

} //1-ый цикл

set\_g[1]=random()\*m;

for i=2..g{ 2-ой цикл

for j=1..i-1{ 3-ий цикл

b.add(set\_g[j]);

}

c=a\b;

ab=new Array(size(c));

x=0;

for j=1..m{ //4-ый цикл

if(c.has(j)){ab[x]=j; x=x+1;}

}

num=random()\*size(c);

set\_g[i]=ab[num];

}

После выполнения этого алгоритма в массиве set\_g будет содержаться g случайных вершин. Сложность этого алгоритма будет равна 2\*m+4+(g-1)\*{сложность тела 2-го цикла}. Тело 2-го цикла имеет линейную сложность относительно m и g, так как в этом теле выполняется 2 цикла: 3-ий цикл, тело которого выполняется не более g-1 раз и имеет одну операцию, и 4-ый цикл, тело которого выполняется не более m раз и имеет две операции. Также в теле 2-го цикла выполняется ещё пять простых операций. Следовательно, функция сложности алгоритма будет квадратичной относительно параметра g. Следовательно, алгоритм генерации множества V1 имеет полиномиальную сложность.

Построим формальный алгоритм проверки того, является ли данное множество из l вершин графа независимым множеством вершин графа:

i=1; flag=true;

while((i<=m)and(flag)){

j=1;

while((j<=m)and(flag)){

if(mat[i][j]==1){

if((i in set\_g) and (j in set\_g)){ flag=false;}

else{j++}

}

else{j=j+1}

}

i=i+1;

}

И внешний, и внутренний циклы выполняются не более m раз, следовательно, алгоритм имеет полиномиальную (квадратичную) сложность. Следовательно, алгоритм проверки того, является ли данное множество независимым множеством, также имеет полиномиальную сложность. Следовательно, первый этап имеет полиномиальную сложность.

На втором этапе производится перебор m-g вершин множества V/V1, с добавлением на каждой итерации текущий вершины в V1. Для каждого из множеств {V1+v} выполняется алгоритм проверки того, является ли это множество независимым множеством. Этот алгоритм имеет квадратичную сложность от m. Следовательно, второй этап имеет полиномиальную (кубическую) сложность от m. Следовательно, задача принадлежит классу NP.

**Доказательство принадлежности к классу NP – полных задач**

Известно, что множество V1⊂V графа G(V,E) является независимым множеством тогда и только тогда, когда его дополнение V2=V\V1 является вершинным покрытием графа G. Пусть Q – следующая задача: дан граф G(V,E), V=m. Найти минимальное вершинное покрытие данного графа. Эта задача является NP-полной. Задача Q сводится к задаче о независимом множестве за полиномиальное время, так как для решения задачи о независимом множестве можно сначала использовать алгоритм решения задачи Q, а затем вычислить разность множества вершин и найденного минимального вершинного покрытия (это выполняется за полиномиальное время), что и будет составлять максимальное независимое множество. Таким образом, существует алгоритм сведения одной задачи к другой со сложностью O(m), где m – количество вершин графа.

**Определить, является ли задача NP-трудной и  NP-полной в сильном смысле (существует ли для ее решения псевдополиномиальный алгоритм).**

Исходная задача является NP-трудной, так как соответствующая ей задача принятия решения (поиска независимого множества графа) является NP-полной. Это следует из предыдущего пункта о сводимости данной задачи к задаче поиска минимального вершинного покрытия. Данные задачи являются частными случаями задач поиска независимого множества и поиска вершинного покрытия графа. А эти задачи в свою очередь являются np-полными.

Но для полных графов исходная задача имеет полиномиальную сложность, так как проверка того, является ли граф полным, выполняется за полиномиальное (квадратичное) время относительно числа вершин в графе. Если граф с m вершинами полный, то в нём максимальным независимым множеством будет любая m вершин. Это следует из того, что в полном графе каждая из вершин смежна со всеми остальными вершинами. Следовательно, для полных графов задача может быть решена точно и за полиномиальное время. Следовательно, существует псевдополиномиальный алгоритм решения исходной задачи. Следовательно, данная задача не является np-полной в сильном смысле.

**Приближённые методы решения задачи и алгоритм полного перебора.**

В ходе работы было реализовано 3 приближенных и 1 точный метод (полный перебор) решения задачи.

**1 Точный метод решения – Полный перебор.**

*Оценка сложности данного алгоритма O(2^n)*

Точный метод решения задачи вершинного покрытия состоит в полном переборе всех возможных вариантов и состоит из следующих шагов:

1. Создается массив из m вершин в котором будет хранится информация информация о выбранных вершинах в процессе построения независимого множества.
2. Создается список вершин (или независимое множество) – он изначально пустой.
3. В ходе цикла от 1 до m запускаем функцию построения независимого множества из определенного числа вершин (от 1 до m). Данная функция рекурсивно строит все возможные независимые множества размера от 1 до m. Эта функция возвращает true или false в зависимости от того, является ли данное множество независимым множеством вершин графа. Цикл останавливается, в том случае если было найдено решение задачи – максимальное независимое множество.

**2 Приближенный метод решения – Жадный алгоритм.**

*Оценка сложности данного алгоритма O(n^4)*

Данный алгоритм является эвристикой.

1. Создается список вершин (независимое множество) – он изначально пустой.
2. Выбирается случайное ребро, которое ещё не было покрыто.
3. Из этого ребра выбирается одна вершина.
4. Вершина добавляется в независимое множество (список вершин из 1 шага)
5. Происходит удаление всех смежных с добавленной вершиной вершин.
6. Если число оставшихся вершин не равно нулю, то в таком случае переходим в шаг 2; Если оно равно нулю, это значит что мы построили максимальное независимое множество.

**3 Приближенный метод решения – *Advanced Vertex Support Algorithm*.**

*Оценка сложности данного алгоритма O(EVlog(v))*

Данный алгоритм является эвристикой.

1. Создается список вершин (независимое множество) – он изначально пустой.
2. Вычисляется степень каждой вершины.
3. Вычисляется значение «поддержки» (support value)  каждой вершины. Поддержка вершины – это сумма степеней всех смежной с данной вершиной вершин.
4. Создание списка вершин с наименьшим значением поддержки.
5. Создание списка вершин смежных с вершинами из списка созданного в четвертом пункте.
6. Из списка созданного на шаге 5, выбирается вершина, которая имеет наибольшее значение поддержки для вершин, созданных на шаге 4.
7. Вершина добавляется в независимое множество (список вершин из 1 шага)
8. Происходит удаление всех вершин смежных с добавленной вершиной.
9. Если число оставшихся вершин не равно нулю, то в таком случае переходим в шаг 2; Если оно равно нулю, это значит что мы построили минимальное вершинное покрытие.

**4 Приближенный метод решения – Генетический алгоритм.**

*Оценка сложности данного алгоритма O(n^2)*

Данный алгоритм является биоинспирированным стохастическим методом.

1. Формирование случайной начальной популяции особей.
2. Оценивание приспособленности каждой особи – *приспособленность особи = число покрытых ребер - число непокрытых ребер \* общее число ребер.* То есть чем меньше вершин в покрытии тем выше приспособленность особи.
3. Селекция – выбирается 20% лучших по значению приспособленности особей (родителей) и записываются в список. Выбирается 5% случайных особей (родителей)  и записывается в список.
4. Скрещивание – происходит перемешивание созданного ранее списка родителей. Далее происходит попарное скрещивание родителей. Число потомков для каждой пары равно восьми. Затем происходит мутация.
5. Повторение шагов начиная с шага 2 или выход по заданному условию.

На выходе мы получаем минимальное вершинное покрытие в виде наилучшей особи последнего поколения. Выкинем данные вершины из общего множества вершин и получим максимальное независимое множество графа.

Особь популяции состоит из хромосомы. Хромосомы (альтернативные решения задачи) состоят и m генов. Кодировка генов бинарная. Например, хромосома 1001 означает, что вершинное покрытие состоит из вершин 1 и 4.

**Результаты работы программы:**

На рисунках представлен результат работы программы – найденное минимальное вершинное покрытие, отрисовка графа, матрица смежности графа.

Точное решение выделено зеленым цветом на графе.



Рисунок 1 – Результат работы программы (Пример 1).

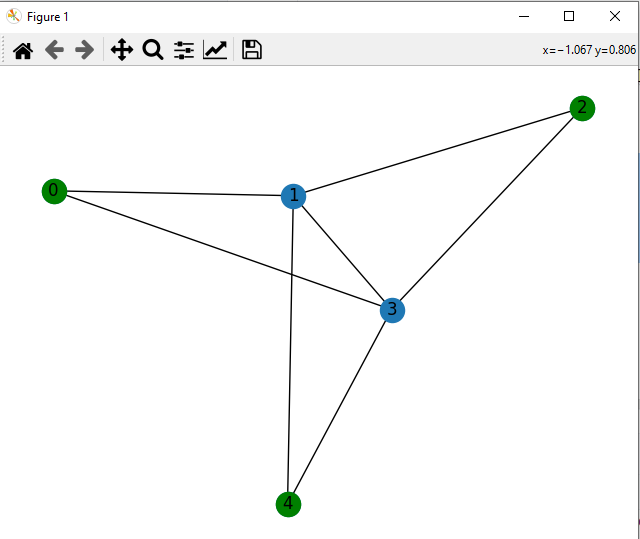


Рисунок 2 – сгенерированный граф с решением задачи (Пример 1).

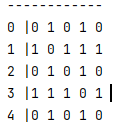


Рисунок 3 – матрица смежности сгенерированного графа (Пример 1).



Рисунок 4 – Результат работы программы (Пример 2).

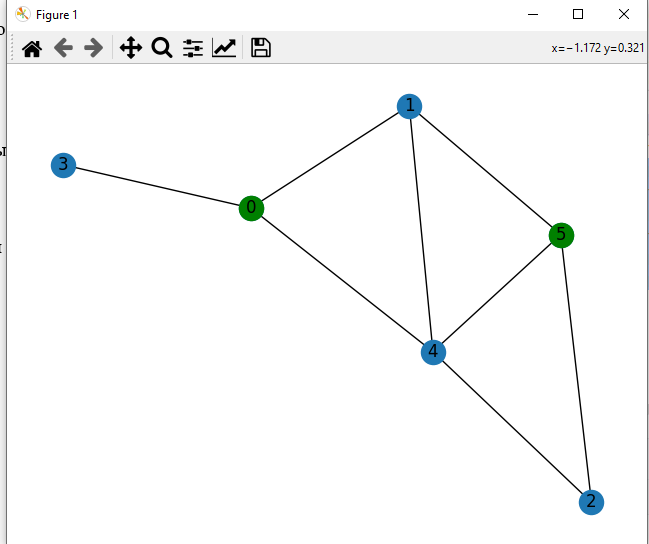


Рисунок 5 – сгенерированный граф с решением задачи (Пример 2).

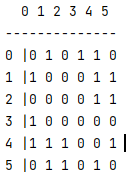


Рисунок 6 – матрица смежности сгенерированного графа (Пример 2).



Рисунок 7 – Результат работы программы (Пример 3).

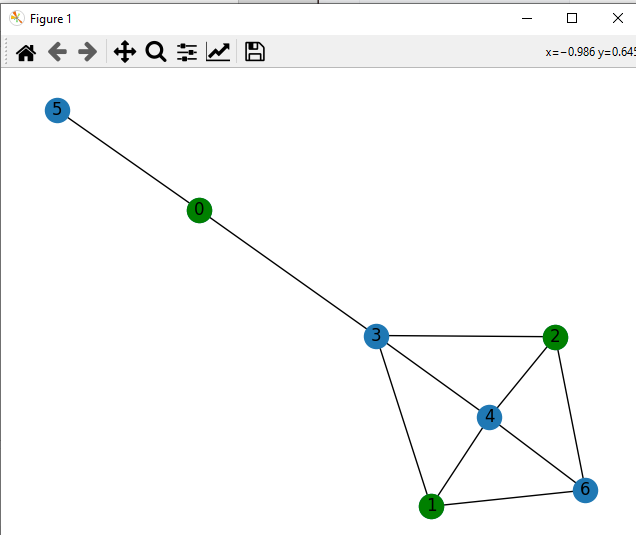


Рисунок 8 – сгенерированный граф с решением задачи (Пример 3).

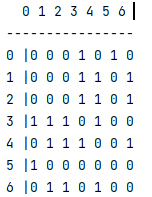


Рисунок 9 – матрица смежности сгенерированного графа (Пример 3).



Рисунок 10 – Результат работы программы (Пример 4).

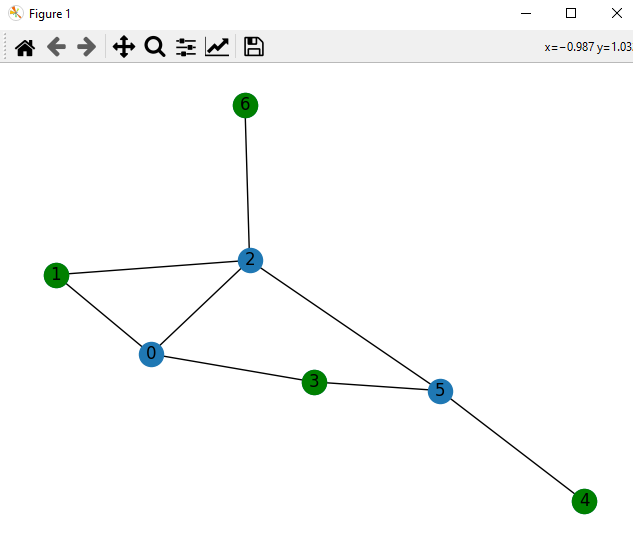


Рисунок 11 – сгенерированный граф с решением задачи (Пример 4).

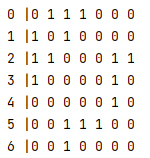


Рисунок 12 – матрица смежности сгенерированного графа (Пример 2).

**Текст программы:**

import numpy as np  
import networkx as nx  
import matplotlib.pyplot as plt  
from random import choice, random  
import random  
import warnings  
from typing import List, Tuple, Set  
  
warnings.filterwarnings("ignore")  
  
class fullSearch:  
 def \_\_init\_\_(self,m):  
 self.matrix = GraphMatrix(m)  
 self.full\_search()  
 print(self.solution)  
 show\_solution(self.matrix.matrix, self.solution)  
 self.matrix.print\_matrix()  
  
 def is\_cover(self) -> bool:  
 covers = True  
 m = len(self.matrix.matrix)  
  
 for i in range(m):  
 for j in range(i + 1, m, 1):  
 if self.matrix.matrix[i][j]:  
 if (i in self.solution and j in self.solution):  
 covers = False  
 break  
 if not covers:  
 break  
  
 return covers  
  
  
 def full\_search(self):  
 m = len(self.matrix.matrix)  
 visited = [False] \* m  
 self.solution = []  
 def search(cover: List[int], n: int) -> bool:  
 if n == 0:  
 return self.is\_cover()  
 else:  
 for i in range(m):  
 if not visited[i]:  
 visited[i] = True  
 cover.append(i)  
 if search(cover, n - 1):  
 return True  
 else:  
 visited[i] = False  
 cover.pop()  
 return False  
  
 for n in range(m-1, 1,-1):  
 if search(self.solution, n):  
 break  
  
 return self.solution  
  
  
class simpleGreed:  
 def \_\_init\_\_(self,m):  
 self.matrix = GraphMatrix(m)  
 self.simple\_greedy()  
 print(self.solution)  
 show\_solution(self.matrix.matrix, self.solution)  
 self.matrix.print\_matrix()  
 def simple\_greedy(self):  
 self.vertexes = [i for i in range(len(self.matrix.matrix))]  
 self.solution = []  
 def remove\_vertexes(vert):  
 self.vertexes = [ v for v in self.vertexes if not self.matrix.matrix[v][vert]]  
 while len(self.vertexes) != 0:  
 vert = choice(self.vertexes)  
 self.vertexes.remove(vert)  
 self.solution.append(vert)  
 remove\_vertexes(vert)  
  
 return self.solution  
  
  
class GraphMatrix:  
 def \_\_init\_\_(self,m):  
 self.matrix = self.generate\_matrix(m)  
 def generate\_matrix(self,m):  
 *"""Генерация случайной матрицы смежности графа."""* self.matrix = [[False] \* m for \_ in range(m)]  
  
 for i in range(m):  
 for j in range(i + 1, m):  
 if choice((True, False)):  
 self.matrix[i][j] = True  
 self.matrix[j][i] = True  
  
 return self.matrix  
 def edges\_to\_ndarray(edges: List[Tuple[int, int]]):  
 num\_edges = len(edges)  
  
 res = np.ndarray(shape=(num\_edges, 2), dtype=int)  
 for i, (v1, v2) in enumerate(edges):  
 res[i, 0] = v1  
 res[i, 1] = v2  
  
 return res  
 def print\_matrix(self):  
 m = len(self.matrix)  
 width = len(str(m)) + 1  
  
 text = [f'{"":^{width}}{"".join(f"{i:^{width}}" for i in range(m))}']  
 text.append('-' \* len(text[0]))  
 for i, row in enumerate(self.matrix):  
 text.append(f'{f"{i}":^{width}}|{"".join(f"{1 if self.matrix[i][j] else 0:^{width}}" for j in range(m))}')  
  
 print('\n'.join(text))  
  
  
class AdvancedVertexSupportAlgorithm:  
 def \_\_init\_\_(self,m):  
 self.matrix = GraphMatrix(m)  
 self.avsa()  
 print(self.solution)  
 show\_solution(self.matrix.matrix, self.solution)  
 self.matrix.print\_matrix()  
 def avsa(self) -> List[int]:  
 matrix = self.matrix.matrix  
 m = len(matrix)  
 edges = matrix\_to\_edges(matrix)  
 self.solution = []  
 self.verticles = [i for i in range(m)]  
  
  
 def degree(edges: List[Tuple[int, int]], v: int) -> int:  
 degree = 0  
 for v1, v2 in edges:  
 if v1 == v or v2 == v:  
 degree += 1  
 return degree  
  
 def support(edges: List[Tuple[int, int]], degrees: List[int], v: int) -> int:  
 res = 0  
 for (v1, v2) in edges:  
 if v1 == v:  
 res += degrees[v1]  
 elif v2 == v:  
 res += degrees[v2]  
  
 return res  
  
 def min\_elements(support\_values: List[int]) -> List:  
 *"""Нахождение вершинын наименьшим значением поддержки."""* res = [None] \* len(support\_values)  
 l = [value for value in support\_values if value is not None]  
 if l:  
 m = min([value for value in support\_values if value is not None])  
 for i, value in enumerate(support\_values):  
 if value is not None:  
 if value == m:  
 res[i] = value  
  
 return res  
  
 def find\_max\_neighbor(min\_support: List[int], edges: List[Tuple[int, int]]) -> int:  
 *"""Нахождение соседа с максимальным значением поддержки"""* max\_neighbor = None  
 max\_support = None  
  
 for v1 in self.verticles:  
 sup1 = 0  
 for v2, sup2 in enumerate(min\_support):  
 if sup2 is not None:  
 if (v1, v2) in edges or (v2, v1) in edges:  
 sup1 += sup2  
  
 if max\_neighbor is None:  
 max\_neighbor = v1  
 max\_support = sup1  
 elif max\_support < sup1:  
 max\_neighbor = v1  
 max\_support = sup1  
  
 return max\_neighbor  
  
 while len(self.verticles) != 0:  
 degrees = [degree(edges, v) for v in range(m)]  
 support\_values = [support(edges, degrees, v) if degrees[v] != 0 else None for v in range(m)]  
 min\_support = min\_elements(support\_values)  
 neighbor = find\_max\_neighbor(min\_support, edges)  
 self.solution.append(neighbor)  
 self.verticles = [v for v in self.verticles if (neighbor,v) not in edges and (v,neighbor) not in edges]  
 self.verticles.remove(neighbor)  
  
  
def edges\_to\_ndarray(edges: List[Tuple[int, int]]):  
 num\_edges = len(edges)  
  
 res = np.ndarray(shape=(num\_edges, 2), dtype=int)  
 for i, (v1, v2) in enumerate(edges):  
 res[i, 0] = v1  
 res[i, 1] = v2  
  
 return res  
  
def genetic\_alg(m):  
 matrix = GraphMatrix(m)  
 m = len(matrix.matrix)  
 edges = matrix\_to\_edges(matrix.matrix)  
 num\_edges = len(edges)  
 edges = edges\_to\_ndarray(edges)  
  
 class Agent:  
 def \_\_init\_\_(self, generation, dna, mutation\_rate):  
 self.generation = generation  
 self.dna = dna  
 self.mutation\_rate = mutation\_rate  
 self.fitness = 0  
  
 def mutate(self):  
 *"""Оператор мутации"""* self.dna = self.dna.copy()  
 mutation\_mask = np.random.rand(m) < self.mutation\_rate / m  
 self.dna[mutation\_mask] = 1 - self.dna[mutation\_mask] *# перестановка* return self  
  
 def crossover(self, other\_agent, offspring: int):  
 *"""Скрещивание особей"""* selection = np.random.choice([0, 1], size=(m)).astype(np.bool\_)  
 dna = np.choose(selection, [self.dna, other\_agent.dna])  
 return [Agent(self.generation + 1, dna, self.mutation\_rate)  
 for i in range(offspring)]  
  
 def update\_fitness(self):  
 *"""Приспособленность особи = сумма чисел - число непокрытых ребер \* число ребер"""* vert\_cover\_fitness = np.full([num\_edges], -num\_edges)  
 mask = (self.dna[edges[:, 0]] | self.dna[edges[:, 1]]).astype(bool)  
 vert\_cover\_fitness[mask] = 1.0  
 self.fitness = np.sum(vert\_cover\_fitness) - np.sum(self.dna)  
 return self.fitness  
  
 class GA:  
 def solve(self, population\_size=30, mutation\_rate=1):  
 self.population\_fitness = []  
 self.populatin\_size = population\_size  
  
 initial\_dna = np.ones(m, dtype=np.int\_)  
 population = [Agent(1, initial\_dna, mutation\_rate) for \_ in range(self.populatin\_size)]  
 population\_fitness = [agent.update\_fitness() for agent in population]  
 self.population\_fitness.append(np.max(population\_fitness))  
  
 result = self.run(population)  
  
 result.sort(key=lambda ag: ag.fitness, reverse=True)  
 return result[0]  
  
 def selection(self, population):  
 *"""  
 Выбор 20% лучших по значению приспособленности  
 и выбор 5% случайных особей  
 """* top\_n = round(self.populatin\_size \* 0.2)  
 lucky\_losers = round(self.populatin\_size \* 0.05)  
 winners = sorted(population, key=lambda ag: ag.fitness, reverse=True)  
 return winners[:top\_n] + random.sample(winners[top\_n:], lucky\_losers)  
  
 def crossover(self, parents):  
 *"""Скрещивание особей"""* offspring = round(self.populatin\_size / (len(parents) / 2))  
 random.shuffle(parents)  
 children = []  
 for i in range(0, len(parents) // 2):  
 children += parents[i \* 2].crossover(parents[i \* 2 + 1], offspring)  
 return children  
  
 def run(self, population):  
 generation = 0  
 delta\_fitness = 2  
  
 while delta\_fitness > 1 and generation < 100:  
 parents = self.selection(population) *# селекция особей* next\_gen = self.crossover(parents) *# скрещивание* mut\_gen = [agent.mutate() for agent in next\_gen] *# мутирование нового поколения* population\_fitness = [agent.update\_fitness() for agent in mut\_gen] *# вычисление приспособленности* if len(self.population\_fitness) >= 50:  
 delta\_fitness = abs(np.sum(np.diff(self.population\_fitness[-50:])))  
 population = mut\_gen  
  
 generation += 1  
 self.population\_fitness.append(np.max(population\_fitness))  
 return population  
  
 best\_agent = GA().solve()  
 result = [i for i, gene in enumerate(best\_agent.dna) if gene == 1]  
 result = [i for i in range(len(matrix.matrix)) if i not in result]  
 print(result)  
 show\_solution(matrix.matrix,result)  
 print(matrix.print\_matrix(),sep='\n')  
  
  
def show\_solution(graph: List[List[bool]], cover: List[int]):  
 edges = matrix\_to\_edges(graph)  
 m = len(graph)  
  
 g = nx.Graph()  
 g.add\_nodes\_from(range(m))  
 g.add\_edges\_from(edges)  
  
 pos = nx.spring\_layout(g)  
 nx.draw(g,  
 pos=pos,  
 with\_labels=True)  
 nx.draw\_networkx\_nodes(g,  
 pos=pos,  
 nodelist=[v for v in range(m) if v in cover],  
 node\_color='g')  
 plt.show()  
  
def matrix\_to\_edges(matrix: List[List[bool]]) -> List[Tuple[int, int]]:  
 *"""Конвертация матрицы смежности вершин в список ребер"""* m = len(matrix)  
 edges = []  
  
 for i in range(m):  
 for j in range(i + 1, m, 1):  
 if matrix[i][j]:  
 edges.append((i, j))  
  
 return edges  
  
print("Точный алгоритм")  
f = fullSearch(5)  
print("Жадный алгоритм")  
f = simpleGreed(6)  
print("AVSA")  
f = AdvancedVertexSupportAlgorithm(7)  
print("Генетический алгоритм")  
print(genetic\_alg(7))