**Міністерство освіти і науки України**

**Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"**

**Факультет інформатики та обчислювальної техніки**

**Кафедра інформатики та програмної інженерії**

**Звіт**

з лабораторної роботи № 5 з дисципліни

«Проектування алгоритмів»

„**Проектування і аналіз алгоритмів для вирішення NP-складних задач ч.2**”





Київ 2022

Зміст

[**1**](#_heading=h.30j0zll) **Мета лабораторної роботи 3**

[**2**](#_heading=h.1fob9te) **Завдання 4**

[**3**](#_heading=h.3znysh7) **Виконання 10**

[3.1](#_heading=h.2et92p0) Покроковий алгоритм 10

[3.2](#_heading=h.tyjcwt) Програмна реалізація алгоритму 10

[*3.2.1*](#_heading=h.3dy6vkm) *Вихідний код 10*

[*3.2.2*](#_heading=h.1t3h5sf) *Приклади роботи 10*

[3.3](#_heading=h.4d34og8) Тестування алгоритму 11

[**Висновок 12**](#_heading=h.2s8eyo1)

[**Критерії оцінювання 13**](#_heading=h.17dp8vu)

# Мета лабораторної роботи

Мета роботи – вивчити основні підходи розробки метаеврестичних алгоритмів для типових прикладних задач. Опрацювати методологію підбору прийнятних параметрів алгоритму.

# Завдання

Згідно варіанту, формалізувати алгоритм вирішення задачі відповідно загальної методології.

Записати розроблений алгоритм у покроковому вигляді. З достатнім степенем деталізації.

Виконати його програмну реалізацію на будь-якій мові програмування.

Перелік задач наведено у таблиці 2.1.

Перелік алгоритмів і досліджуваних параметрів у таблиці 2.2.

Задача і алгоритм наведені в таблиці 2.3.

Змінюючи параметри алгоритму, визначити кращі вхідні параметри алгоритму. Для цього необхідно:

* обрати критерій зупинки алгоритму (кількість ітерацій або значення ЦФ);
* зафіксувати усі параметри крім одного і змінювати цей параметр, поки не буде досягнуто пікової ефективності;
* після цього параметр фіксується і змінюються інші параметри;
* далі повторюємо процедуру спочатку, з першого зафіксованого параметру;
* зупиняємось коли будуть знайдені оптимальні параметри для даної задачі або встановлена залежність одних параметрів від інших.

Зробити узагальнений висновок в якому обов’язково описати залежність якості розв’язку від вхідних параметрів.

Таблиця 2.1 – Прикладні задачі

| **№** | **Задача** |
| --- | --- |
| 1 | **Задача про рюкзак** (місткість P=500, 100 предметів, цінність предметів від 2 до 30 (випадкова), вага від 1 до 20 (випадкова)). Для заданої множини предметів, кожен з яких має вагу і цінність, визначити яку кількість кожного з предметів слід взяти, так, щоб сумарна вага не перевищувала задану, а сумарна цінність була максимальною.  Задача часто виникає при розподілі ресурсів, коли наявні фінансові обмеження, і вивчається в таких областях, як комбінаторика, інформатика, теорія складності, криптографія, прикладна математика. |
| 2 | **Задача комівояжера** (300 вершин, відстань між вершинами випадкова від 5 до 150) полягає у знаходженні найвигіднішого маршруту, що проходить через вказані міста хоча б по одному разу. В умовах завдання вказуються критерій вигідності маршруту (найкоротший, найдешевший, сукупний критерій тощо) і відповідні матриці відстаней, вартості тощо. Зазвичай задано, що маршрут повинен проходити через кожне місто тільки один раз, в такому випадку розв'язок знаходиться серед гамільтонових циклів.  **Розглядається симетричний, асиметричний та змішаний варіанти.**  В загальному випадку, асиметрична задача комівояжера відрізняється тим, що ребра між вершинами можуть мати різну вагу в залежності від напряму, тобто, задача моделюється орієнтованим графом. Таким чином, окрім ваги ребер графа, слід також зважати і на те, в якому напрямку знаходяться ребра.  У випадку симетричної задачі всі пари ребер між одними й тими самими вершинами мають однакову вагу.  У випадку реальних міст може бути як симетричною, так і асиметричною в залежності від тривалості або довжини маршрутів і напряму руху.  Застосування:   * доставка товарів (в цьому випадку може бути більш доречна постановка транспортної задачі - доставка в кілька магазинів з декількох складів); * доставка води; * моніторинг об'єктів; * поповнення банкоматів готівкою; * збір співробітників для доставки вахтовим методом. |
| 3 | **Розфарбовування графа** (300 вершин, степінь вершини не більше 30, але не менше 2) – називають таке приписування кольорів (або натуральних чисел) його вершинам, що ніякі дві суміжні вершини не набувають однакового кольору. Найменшу можливу кількість кольорів у розфарбуванні називають хроматичне число.  Застосування:   * розкладу для освітніх установ; * розкладу в спорті; * планування зустрічей, зборів, інтерв'ю; * розклади транспорту, в тому числі - авіатранспорту; * розкладу для комунальних служб; |
| 4 | **Задача вершинного покриття** (300 вершин, степінь вершини не більше 30, але не менше 2)**.** Вершинне покриття для неорієнтованого графа G = (V, E) - це множина його вершин S, така, що, у кожного ребра графа хоча б один з кінців входить в вершину з S.  Задача вершинного покриттяполягає в пошуку вершинного покриття найменшого розміру для заданого графа (цей розмір називається числом вершинного покриття графа).  На вході: Граф G = (V, E).  Результат: множина C ⊆ V - найменше вершинне покриття графа G.    Застосування:   * розміщення пунктів обслуговування; * призначення екіпажів на транспорт; * проектування інтегральних схем і конвеєрних ліній. |
| 5 | **Задача про кліку** (300 вершин, степінь вершини не більше 30, але не менше 2)**.** Клікою в неорієнтованому графі називається підмножина вершин, кожні дві з яких з'єднані ребром графа. Іншими словами, це повний підграф первісного графа. Розмір кліки визначається як число вершин в ній.  Задача про кліку існує у двох варіантах: у **задачі розпізнавання** потрібно визначити, чи існує в заданому графі G кліка розміру k, тоді як в **обчислювальному варіанті** потрібно знайти в заданому графі G кліку максимального розміру або всі максимальні кліки (такі, що не можна збільшити).  Застосування:   * біоінформатика; * електротехніка; |
| 6 | **Задача про найкоротший шлях** (300 вершин, відстань між вершинами випадкова від 5 до 150, степінь вершини не більше 10, але не менше 1) - задача пошуку найкоротшого шляху (ланцюга) між двома точками (вершинами) на графі, в якій мінімізується сума ваг ребер, що складають шлях.  Важливість задачі визначається її різними практичними застосуваннями. Наприклад, в GPS-навігаторах здійснюється пошук найкоротшого шляху між точкою відправлення і точкою призначення. Як вершин виступають перехрестя, а дороги є ребрами, які лежать між ними. Якщо сума довжин доріг між перехрестями мінімальна, тоді знайдений шлях найкоротший. |

Таблиця 2.2 – Варіанти алгоритмів і досліджувані параметри

| **№** | **Алгоритми і досліджувані параметри** |
| --- | --- |
| 1 | **Генетичний алгоритм:**   * оператор схрещування (мінімум 3); * мутація (мінімум 2); * оператор локального покращення (мінімум 2). |
| 2 | **Мурашиний алгоритм**:   * α; * β; * ρ; * Lmin; * кількість мурах М і їх типи (елітні, тощо…); * маршрути з однієї чи різних вершин. |
| 3 | **Бджолиний алгоритм:**   * кількість ділянок; * кількість бджіл (фуражирів і розвідників). |

Таблиця 2.3 – Варіанти задач і алгоритмів

| **№** | **Задачі і алгоритми** |
| --- | --- |
| 1 | Задача про рюкзак + Генетичний алгоритм |
| 2 | Задача про рюкзак + Бджолиний алгоритм |
| 3 | Задача комівояжера (асиметрична мережа) + Генетичний алгоритм |
| 4 | Задача комівояжера (симетрична мережа) + Генетичний алгоритм |
| 5 | Задача комівояжера (змішана мережа) + Генетичний алгоритм |
| 6 | Задача комівояжера (асиметрична мережа) + Мурашиний алгоритм |
| 7 | Задача комівояжера (симетрична мережа) + Мурашиний алгоритм |
| 8 | Задача комівояжера (змішана мережа) + Мурашиний алгоритм |
| 9 | Задача вершинного покриття + Генетичний алгоритм |
| 10 | Задача вершинного покриття + Бджолиний алгоритм |
| 11 | Задача комівояжера (асиметрична мережа) + Бджолиний алгоритм |
| 12 | Задача комівояжера (симетрична мережа) + Бджолиний алгоритм |
| 13 | Задача комівояжера (змішана мережа) + Бджолиний алгоритм |
| 14 | Розфарбовування графа + Генетичний алгоритм |
| 15 | Розфарбовування графа + Бджолиний алгоритм |
| 16 | Задача про кліку (задача розпізнавання) + Генетичний алгоритм |
| 17 | Задача про кліку (задача розпізнавання) + Бджолиний алгоритм |
| 18 | Задача про кліку (обчислювальна задача) + Генетичний алгоритм |
| 19 | Задача про кліку (обчислювальна задача) + Бджолиний алгоритм |
| 20 | Задача про найкоротший шлях + Генетичний алгоритм |
| 21 | Задача про найкоротший шлях + Мурашиний алгоритм |
| 22 | Задача про найкоротший шлях + Бджолиний алгоритм |
| 23 | Задача про рюкзак + Генетичний алгоритм |
| 24 | Задача про рюкзак + Бджолиний алгоритм |
| 25 | Задача комівояжера (асиметрична мережа) + Генетичний алгоритм |
| 26 | Задача комівояжера (симетрична мережа) + Генетичний алгоритм |
| 27 | Задача комівояжера (змішана мережа) + Генетичний алгоритм |
| 28 | Задача комівояжера (асиметрична мережа) + Мурашиний алгоритм |
| 29 | Задача комівояжера (симетрична мережа) + Мурашиний алгоритм |
| 30 | Задача комівояжера (змішана мережа) + Мурашиний алгоритм |

# Виконання

## Опис роботи генетичного алгоритму

На початку еволюції створюється набір предметів, що можуть знаходитися в рюкзаку або ні. Далі створюється випадковий набір розв’язків задач, розмір якого – PopulationSize – є параметром, що можна змінювати. Потім набір розв’язків починає еволюцію в основному циклі з 1000 ітерацій, параметрами якого є pointsCount, mutationsCount та improveCount.

Одна ітерація основного циклу включає в себе такі кроки:

1. Проведення “турніру” – обираються два випадкові розв’язки задачі з набору, та з них обирається той, що має більше значення вартості предметів у рюкзаку. Це проводиться двічі, щоб отримати двох “переможців” для наступного етапу.
2. Кросинговер – обираються pointsCount випадкових чисел зі ста, далі по цих точках відбувається схрещування таким чином, як показано на рис. 3.1.

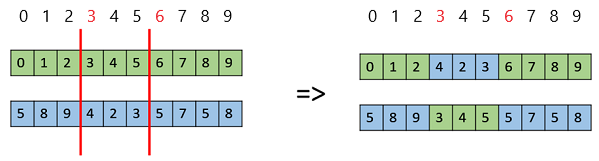


Рисунок 3.1 – схрещування розв’язків до задачі (індекси точок поділу позначені червоним)

1. Мутація – випадкова позиція в отриманому розв’язку “перевертається” з 1 на 0 або з 0 на 1, якщо це не призведе до перегруження рюкзака. Це повторюється mutationsCount разів.
2. Локальне покращення – мутація, але викладає предмет із рюкзака лише якщо він перегружений.
3. Оновлення набору розв’язків – обирається розв’язок задачі, що має найменшу загальну вартість предметів у рюкзаку, та замінюється на розв’язок, отриманий у п. 1-4

Кожні 20 ітерацій циклу записується результат – середня вага рюкзака в усьому наборі розв’язків.

Програма може працювати у двох режимах:

* Еволюції зі сталими параметрами – всі параметри зафіксовані, результатом роботи програми є графік залежності середньої вартості предметів у рюкзаку від кількості пройдених ітерацій (значення усереднені для кількох різних наборів можливих предметів)
* Еволюції з одним змінним параметром – один із параметрів задається як проміжок, результатом роботи є три графіки залежності середньої вартості предметів від значення змінного параметра (для 400, 700 та 1000 ітерацій циклу).

## Покроковий алгоритм

* + 1. Основний цикл та запис результатів

results <- [[] FOR x = 0, x < 51

ENDFOR]

FOR i = 0, i < instances:

items <- []

for i in range(itemsCount):

items.append(Item.Item())

population <- []

FOR i in range (populationSize):

population.append(Solution.Solution(itemsCount, maxWeight))

population[i].createRandom(itemsSet)

ENDFOR

generateSolutions(itemsCount, populationSize, maxWeight, items)

valueGrowth <- []

FOR bigCycle = 0, bigCycle < 50:

valueGrowth.append(avgSolutionValue(population))

FOR smallCycle = 0, smallCycle < 20:

winner1, winner2 <- doTournaments(population, itemsCount)

splitPoints <- getSplittingPositions(pointsCount, itemsCount)

newSolution <- createChild(winner1, winner2, splitPoints, items)

newSolution.mutate(mutationsCount, items)

newSolution.improve(improveCount, items)

upgradeSet(population, newSolution)

ENDFOR

ENDFOR

valueGrowth.append(avgSolutionValue(population))

FOR j = 0, j < len(valueGrowth):

results[j].append(valueGrowth[j])

ENDFOR

ENDFOR

FOR i = 0, i < len(results):

results[i] <- sum(results[i])/len(results[i])

ENDFOR

* + 1. function doTournaments()

FUNCTION doTournaments(solutionsSet, itemsCount):

contestants <- random.sample(solutionsSet, 4)

IF contestants[0].value > contestants[1].value:

winner1 <- contestants[0]

ELSE:

winner1 <- contestants[1]

ENDIF

IF contestants[2].value > contestants[3].value:

winner2 <- contestants[2]

ELSE:

winner2 <- contestants[3]

ENDIF

RETURN winner1, winner2

ENDFUNCTION

* + 1. function getSplittingPositions()

FUNCTION getSplittingPositions(pointsCount, itemsCount):

bannedPoints <- []

chosenPoints <- []

newPoint <- random.randint(1, itemsCount-1)

FOR i = 0, i < pointsCount:

WHILE newPoint in bannedPoints:

newPoint <- random.randint(1, itemsCount-1)

ENDWHILE

chosenPoints.append(newPoint)

bannedPoints.append(newPoint)

ENDFOR

chosenPoints.sort()

RETURN [0] + chosenPoints + [itemsCount]

ENDFUNCTION

* + 1. function createChild()

FUNCTION createChild(parent1, parent2, cuttingPoints, itemsSet):

child1 <- Solution.Solution(parent1.size, parent1.maxWeight)

child2 <- Solution.Solution(parent1.size, parent1.maxWeight)

child1Set <- []

child2Set <- []

switch <- 0

FOR i = 0, i < len(cuttingPoints)-1:

IF switch = 0:

child1Set.extend(parent1.set[cuttingPoints[i]:cuttingPoints[i+1]])

child2Set.extend(parent2.set[cuttingPoints[i]:cuttingPoints[i+1]])

ELSE:

child2Set.extend(parent1.set[cuttingPoints[i]:cuttingPoints[i+1]])

child1Set.extend(parent2.set[cuttingPoints[i]:cuttingPoints[i+1]])

ENDIF

switch <- 1 - switch

ENDFOR

child1.set <- child1Set

child2.set <- child2Set

IF child1.value > child2.value:

RETURN child1

ELSE:

RETURN child2

ENDIF

ENDFUNCTION

* + 1. method Solution.mutate()

FUNCTION mutate(Solution, mutationsCount, itemSet):

FOR i = 0, i < mutationsCount:

ind <- random.randint(0, Solution.size-1)

Solution.set[ind] <- bool(1-Solution.set[ind])

IF Solution.weight > Solution.maxWeight:

Solution.set[ind] <- bool(1-Solution.set[ind])

ENDIF

ENDFOR

ENDFUNCTION

* + 1. method Solution.improve()

FUNCTION improve(Solution, improveCount, itemSet):

FOR i = 0, i < improveCount:

ind = random.randint(0, self.size-1)

IF self.set[ind] == True and self.weight > self.maxWeight:

self.set[ind] = False

ELSEIF self.set[ind] == False and self.weight + itemSet[i].weight <= self.maxWeight:

self.set[ind] = True

ENDIF

ENDFOR

ENDFUNCTION

* + 1. function upgradeSet()

FUNCTION upgradeSet(solutionsSet, newSolution):

minValue <- solutionsSet[0].value

minInd <- 0

FOR i = 1, i < len(solutionsSet):

IF solutionsSet[i].value < minValue:

minInd <- i

minValue <- solutionsSet[i].value

ENDIF

ENDFOR

solutionsSet[minInd] <- newSolution

ENDFUNCTION

## Програмна реалізація алгоритму

### Вихідний код

Knapsack.py

# capacity P=500, 100 items, value 2-30, weight 1-20

# crossover parameter 25 >= C >= 3, mutation 100 >= M >= 2, local improvement 100 >= L >= 2

# one solution is array of 1's and 0's (added to knapsack or not)

# additional parameters: runs count, population size

import runScenarios

while True:

print('Enter the 5 parameters to a simulation')

print('All have to be natural numbers')

print('a - one number; a:b - range of numbers')

print('only one range per simulation')

print('--------------------------------------')

while True:

PS = input('Population size [10:30]: ') # populationSize

PC = input('Crossover cutting points [3:25]: ') # pointsCount

MC = input('Mutations per iteration [2:100]: ') # mutationsCount

IC = input('Local improvements per iteration [2:100]: ') # improveCount

instances = input('Runs count [1:20]: ')

print('--------------------------------------')

Exception = runScenarios.runSimulation(PS, PC, MC, IC, instances)

if Exception is None:

break

else:

print(Exception)

print('Try to enter the parameters again')

print('--------------------------------------')

proceed = input('input "0" to stop the program or "1" to do another run: ')

if proceed == '0':

break

runScenarios.py

import func

import plotting

def no\_Range(PS, PC, MC, IC, instances):

"""if all specified parameters are constant"""

results = func.run(PS, PC, MC, IC, instances)

plotting.oneEvoResults(results, PS, PC, MC, IC, instances)

def PS\_Range(PS1, PS2, PC, MC, IC, instances):

"""if populationSize is in a range"""

results = []

results400 = []

results700 = []

for i in range(PS1, PS2+1):

print(f'{i}th iteration:')

runResults = func.run(i, PC, MC, IC, instances)

results400.append(runResults[20])

results700.append(runResults[35])

results.append(max(runResults))

plotting.rangeResults([results, results700, results400], PS1, PC, MC, IC, instances, 0, PS2)

def PC\_Range(PS, PC1, PC2, MC, IC, instances):

"""if pointsCount is in a range"""

results = []

results400 = []

results700 = []

for i in range(PC1, PC2+1):

print(f'{i}th iteration:')

runResults = func.run(PS, i, MC, IC, instances)

results400.append(runResults[20])

results700.append(runResults[35])

results.append(max(runResults))

plotting.rangeResults([results, results700, results400], PS, PC1, MC, IC, instances, 1, PC2)

def MC\_Range(PS, PC, MC1, MC2, IC, instances):

"""if mutationsCount is in a range"""

results = []

results400 = []

results700 = []

for i in range(MC1, MC2+1):

print(f'{i}th iteration:')

runResults = func.run(PS, PC, i, IC, instances)

results400.append(runResults[20])

results700.append(runResults[35])

results.append(max(runResults))

plotting.rangeResults([results, results700, results400], PS, PC, MC1, IC, instances, 2, MC2)

def IC\_Range(PS, PC, MC, IC1, IC2, instances):

"""if improveCount is in a range"""

results = []

results400 = []

results700 = []

for i in range(IC1, IC2+1):

print(f'{i}th iteration:')

runResults = func.run(PS, PC, MC, i, instances)

results400.append(runResults[20])

results700.append(runResults[35])

results.append(max(runResults))

plotting.rangeResults([results, results700, results400], PS, PC, MC, IC1, instances, 3, IC2)

def runSimulation(PS, PC, MC, IC, instances):

"""choose a function to run"""

rangeInd = func.findRangeInd(PS, PC, MC, IC, instances)

if rangeInd == -2:

return 'Error: more than one range input'

elif rangeInd == -3:

return 'Error: the "Runs count" value cannot be a range'

elif rangeInd == -1: # no range found

if func.areNumeric(PS, PC, MC, IC, instances):

PS, PC, MC, IC, instances = int(PS), int(PC), int(MC), int(IC), int(instances)

else:

return 'Error: not all inputs are numeric'

if func.validNumbers(PS, PC, MC, IC, instances):

no\_Range(PS, PC, MC, IC, instances)

else:

return "Error: some numbers are too big or too small"

else:

if rangeInd == 0: # PS is a range

PS, rangeEnd = PS.split(':')

elif rangeInd == 1: # PC is a range

PC, rangeEnd = PC.split(':')

elif rangeInd == 2: # MC is a range

MC, rangeEnd = MC.split(':')

elif rangeInd == 3: # IC is a range

IC, rangeEnd = IC.split(':')

if func.areNumeric(PS, PC, MC, IC, instances, rangeEnd):

PS, PC, MC, IC, instances, rangeEnd = int(PS), int(PC), int(MC), int(IC), int(instances), int(rangeEnd)

else:

return 'Error: not all inputs are numeric'

if func.validNumbers(PS, PC, MC, IC, instances, rangeInd, rangeEnd) and [PS, PC, MC, IC][rangeInd] < rangeEnd:

if rangeInd == 0: # PS is a range

PS\_Range(PS, rangeEnd, PC, MC, IC, instances)

elif rangeInd == 1: # PC is a range

PC\_Range(PS, PC, rangeEnd, MC, IC, instances)

elif rangeInd == 2: # MC is a range

MC\_Range(PS, PC, MC, rangeEnd, IC, instances)

elif rangeInd == 3: # IC is a range

IC\_Range(PS, PC, MC, IC, rangeEnd, instances)

else:

return "Error: some numbers are too big or too small"

func.py

import Item

import Solution

import random

import plotting

def areNumeric(\*args):

"""return true only if all strings are numeric"""

for item in args:

if not item.isnumeric():

return False

return True

def findRangeInd(PS, PC, MC, IC, instances):

"""determine which of the parameters is a range"""

rangeInd = -1

if ':' in PS:

rangeInd = 0

if ':' in PC:

if rangeInd != -1:

return -2

else:

rangeInd = 1

if ':' in MC:

if rangeInd != -1:

return -2

else:

rangeInd = 2

if ':' in IC:

if rangeInd != -1:

return -2

else:

rangeInd = 3

if ':' in instances:

return -3

return rangeInd

def validNumbers(PS, PC, MC, IC, instances, rangeInd = None, rangeEnd = None):

"""return true only if all numbers are in plausible ranges"""

if (PS > 9 and PS < 31) and (PC > 2 and PC < 26) and (MC > 1 and MC < 101) and (IC > 1 and IC < 101) and (instances > 0 and instances < 21):

if rangeEnd != None:

if rangeInd == 0 and (rangeEnd > 9 and rangeEnd < 31): #PS

return True

else:

return False

if rangeInd == 1 and (rangeEnd > 2 and rangeEnd < 26): #PC

return True

else:

return False

if rangeInd == 2 and (rangeEnd > 1 and rangeEnd < 101): #MC

return True

else:

return False

if rangeInd == 3 and (rangeEnd > 1 and rangeEnd < 101): #IC

return True

else:

return False

else:

return True

else:

return False

def createItemSet(itemsCount):

"""creates a new set of 100 items"""

itemSet = []

for i in range(itemsCount):

itemSet.append(Item.Item())

return itemSet

def generateSolutions(itemsCount, populationSize, maxWeight, itemsSet):

"""creates random solutions to begin the simulation with"""

solutionsSet = []

for i in range (populationSize):

solutionsSet.append(Solution.Solution(itemsCount, maxWeight))

solutionsSet[i].createRandom(itemsSet)

return solutionsSet

def doTournaments(solutionsSet, itemsCount):

"""do tournamets to select above average samples, from which a new solution is created"""

contestants = random.sample(solutionsSet, 4) #4 random results

if contestants[0].value > contestants[1].value:

winner1 = contestants[0]

else:

winner1 = contestants[1]

if contestants[2].value > contestants[3].value:

winner2 = contestants[2]

else:

winner2 = contestants[3]

return winner1, winner2

def getSplittingPositions(pointsCount, itemsCount):

"""determine split points for crossover"""

bannedPoints = []

chosenPoints = []

newPoint = random.randint(1, itemsCount-1)

for i in range(pointsCount):

while newPoint in bannedPoints:

newPoint = random.randint(1, itemsCount-1)

chosenPoints.append(newPoint)

bannedPoints.append(newPoint) #extend([newPoint-1, newPoint, newPoint+1])

chosenPoints.sort()

return [0] + chosenPoints + [itemsCount]

def createChild(parent1, parent2, cuttingPoints, itemsSet):

"""crossover phase - creating 2 new arrays and choosing the better one"""

child1 = Solution.Solution(parent1.size, parent1.maxWeight)

child2 = Solution.Solution(parent1.size, parent1.maxWeight)

child1Set = []

child2Set = []

switch = 0

for i in range(len(cuttingPoints)-1):

if switch == 0:

child1Set.extend(parent1.set[cuttingPoints[i]:cuttingPoints[i+1]])

child2Set.extend(parent2.set[cuttingPoints[i]:cuttingPoints[i+1]])

else:

child2Set.extend(parent1.set[cuttingPoints[i]:cuttingPoints[i+1]])

child1Set.extend(parent2.set[cuttingPoints[i]:cuttingPoints[i+1]])

switch = 1 - switch

child1.set = child1Set

child2.set = child2Set

child1.evaluate(itemsSet)

child2.evaluate(itemsSet)

if child1.value > child2.value:

return child1

else:

return child2

def upgradeSet(solutionsSet, newSolution):

"""replace the worst solution in the population with the new one"""

minValue = solutionsSet[0].value

minInd = 0

for i in range(1, len(solutionsSet)):

if solutionsSet[i].value < minValue:

minInd = i

minValue = solutionsSet[i].value

solutionsSet[minInd] = newSolution

def getAvgValue(solutionsSet, populationSize):

"""average value in the population"""

totalValue = 0

for solution in solutionsSet:

totalValue += solution.value

return round(totalValue/populationSize\*100)/100

def run(populationSize, pointsCount, mutationsCount, improveCount, instances = 1, itemsCount = 100, maxWeight = 500):

"""main cycles"""

results = [[] for x in range(51)]

for i in range(instances):

items = createItemSet(itemsCount)

population = generateSolutions(itemsCount, populationSize, maxWeight, items)

valueGrowth = []

for bigCycle in range(50):

valueGrowth.append(getAvgValue(population, populationSize))

for smallCycle in range(20):

winner1, winner2 = doTournaments(population, itemsCount)

splitPoints = getSplittingPositions(pointsCount, itemsCount)

newSolution = createChild(winner1, winner2, splitPoints, items)

newSolution.mutate(mutationsCount, items)

newSolution.improve(improveCount, items)

upgradeSet(population, newSolution)

valueGrowth.append(getAvgValue(population, populationSize))

print(f'evolution {i+1} successful with {valueGrowth[-1]} end value')

for j in range(len(valueGrowth)):

results[j].append(valueGrowth[j])

for i in range(len(results)):

results[i] = sum(results[i])/len(results[i])

print(f'run of {instances} evolutions ended with {results[-1]} average end value')

return results

plotting.py

import matplotlib.pyplot as plot

import Solution

#plotting functions

def oneEvoResults(values, PS, PC, MC, IC, instances):

"""show detailed results of a run with multiple item sets but unchanging parameters"""

X = []

for i in range(51):

X.append(i\*20)

plot.plot(X, values)

plot.xlabel('Iterations')

plot.ylabel('Average knapsack value')

plot.title(f'PS = {PS}, PC = {PC}, MC = {MC}, IC = {IC}, avg of {instances}')

print('Close the plot window to end the simulation')

plot.show()

def rangeResults(values, PS, PC, MC, IC, instances, rangeVar, rangeEnd):

"""show results of runs with multiple item sets and different parameters"""

X = []

for i in range([PS, PC, MC, IC][rangeVar], rangeEnd+1):

X.append(i)

plot.plot(X, values[0], color='r', label='1000 iterations')

plot.plot(X, values[1], color='g', label='700 iterations')

plot.plot(X, values[2], color='b', label='400 iterations')

plot.xlabel(f'{["Population size", "Crossover cutting points", "Mutations per iteration", "Local improvements per iteration"][rangeVar]}')

plot.ylabel('Max knapsack value')

if rangeVar == 0: #PS

plot.title(f'PS = {PS}..{rangeEnd}, PC = {PC}, MC = {MC}, IC = {IC}, avg of {instances}')

elif rangeVar == 1: #PC

plot.title(f'PS = {PS}, PC = {PC}..{rangeEnd}, MC = {MC}, IC = {IC}, avg of {instances}')

elif rangeVar == 2: #MC

plot.title(f'PS = {PS}, PC = {PC}, MC = {MC}..{rangeEnd}, IC = {IC}, avg of {instances}')

elif rangeVar == 3: #IC

plot.title(f'PS = {PS}, PC = {PC}, MC = {MC}, IC = {IC}..{rangeEnd}, avg of {instances}')

print('Close the plot window to end the simulation')

plot.legend()

plot.show()

Solution.py

import random

class Solution():

"""An array of bool values that indicate if the i-th item is put into the knapsack"""

def \_\_init\_\_(self, itemsCount, maxWeight):

self.size = itemsCount

self.set = []

for i in range(itemsCount):

self.set.append(False)

self.value = 0

self.weight = 0

self.maxWeight = maxWeight

def createRandom(self, itemSet):

"""Create a random solution"""

for i in range(self.size):

self.set[i] = bool(random.getrandbits(1))

self.evaluate(itemSet)

def mutate(self, mutationsCount, itemSet):

"""apply mutations to a solution"""

for i in range(mutationsCount):

ind = random.randint(0, self.size-1)

if self.set[ind] == True:

self.set[ind] = False

self.weight -= itemSet[ind].weight

self.value -= itemSet[ind].value

elif self.set[ind] == False and self.weight + itemSet[i].weight <= self.maxWeight:

self.set[ind] = True

self.weight += itemSet[ind].weight

self.value += itemSet[ind].value

def improve(self, improveCount, itemSet):

"""apply local improvements to a solution"""

for i in range(improveCount):

ind = random.randint(0, self.size-1)

if self.set[ind] == True and self.weight > self.maxWeight:

self.set[ind] = False

self.weight -= itemSet[ind].weight

self.value -= itemSet[ind].value

elif self.set[ind] == False and self.weight + itemSet[i].weight <= self.maxWeight:

self.set[ind] = True

self.weight += itemSet[ind].weight

self.value += itemSet[ind].value

def evaluate(self, itemSet):

"""determine the value of items in the knapsack, 0 if too much weight"""

self.weight = 0

self.value = 0

for i in range(self.size):

if self.set[i] == True:

self.weight += itemSet[i].weight

self.value += itemSet[i].value

if self.weight > self.maxWeight:

self.value = 0

Item.py

import random

class Item(object):

"""One item that can be put in the knapsack"""

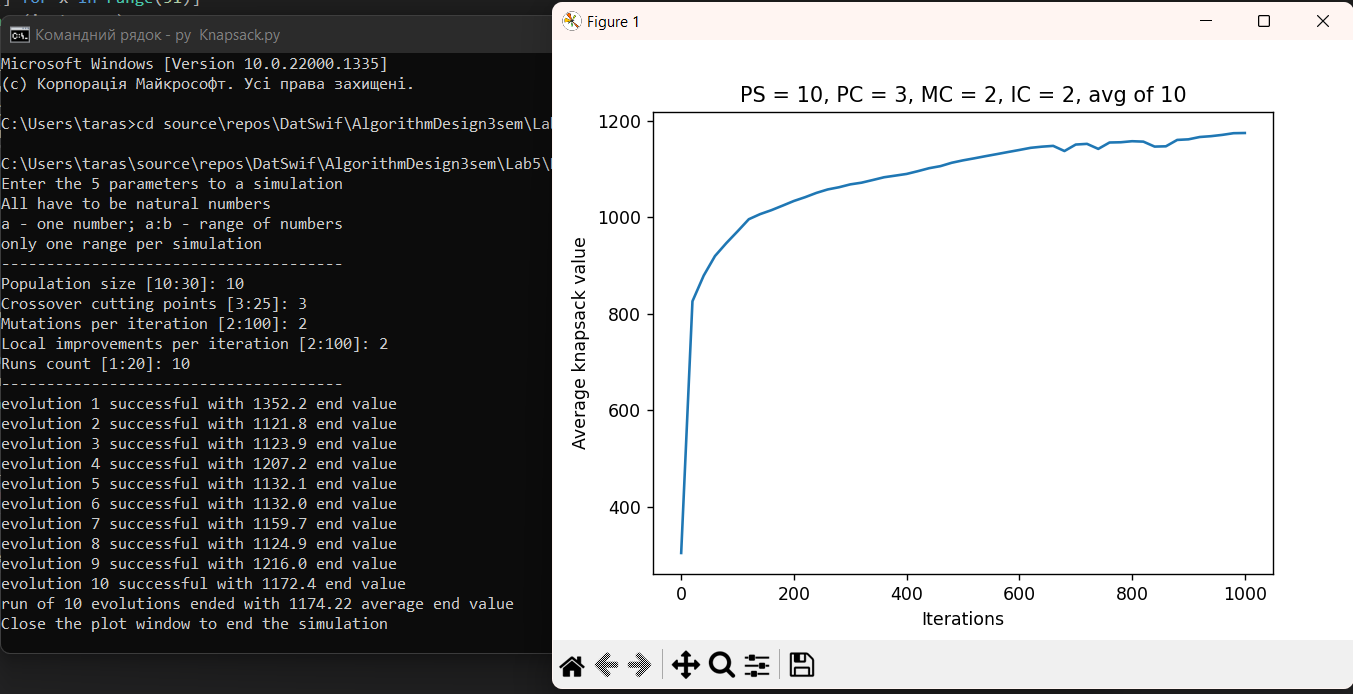
def \_\_init\_\_(self):

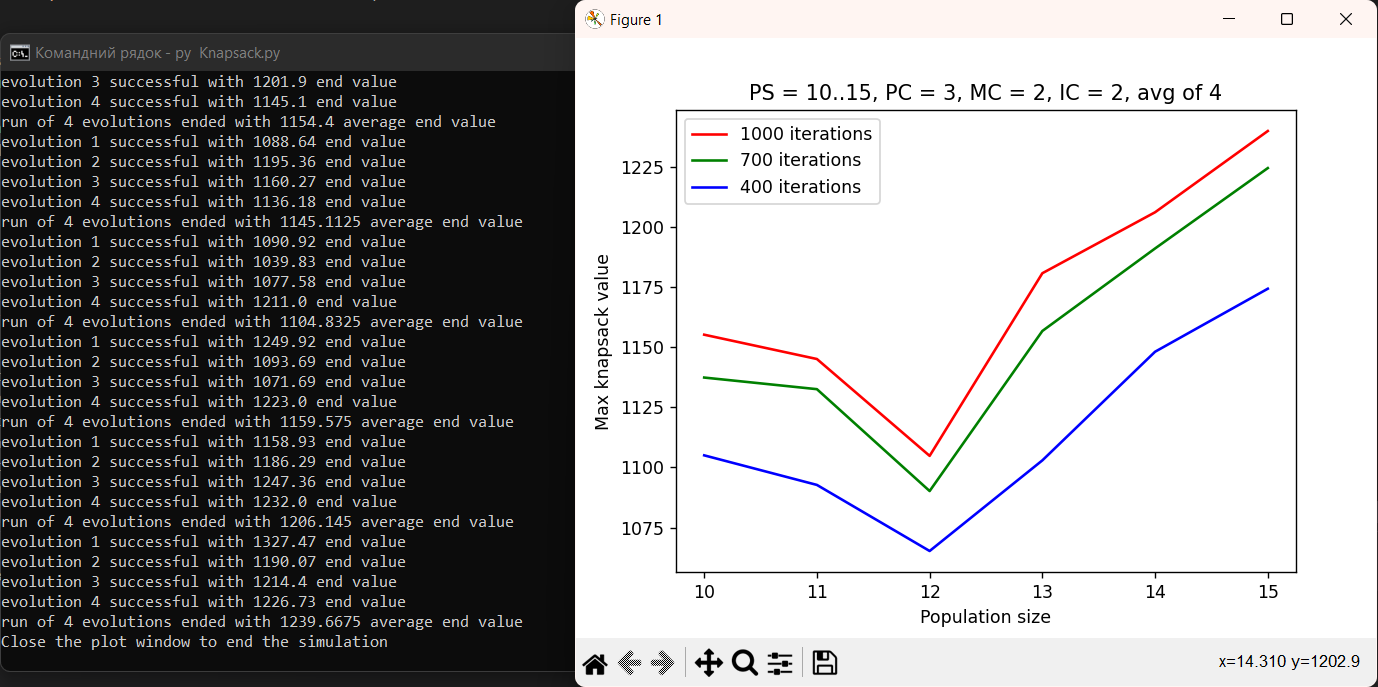
self.weight = random.randint(1, 20)

self.value = random.randint(2, 30)

### Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

Рисунок 3.2 – Робота програми для сталих параметрів

Рисунок 3.3 – Робота програми, параметр PS задано як проміжок

## Тестування алгоритму

### Розмір вибірки (populationSize)

Спочатку встановимо значення параметру PopulationSize, знаходження якого не є завданням роботи. В теорії, надто мала кількість збережених розв’язків зробить процес еволюції хаотичним (більший шанс, що вдалі розв’язки будуть замінюватись невдалими). Спробуємо перевірити це на одній еволюції 10-ти рішень, інші параметри мінімальні (рис. 3.4):

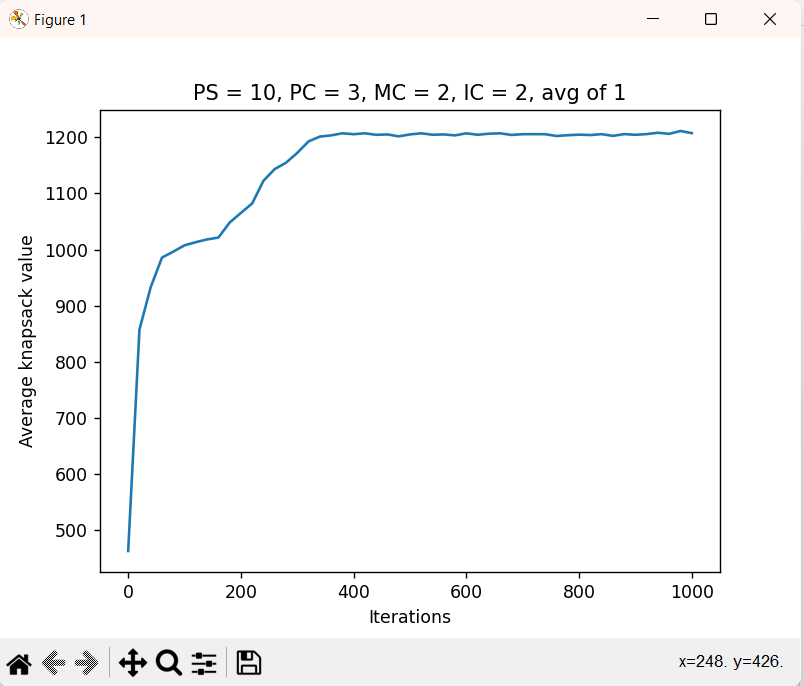


Рисунок 3.4 – Графік зростання вартості є нерівним, але швидко підходить до максимуму

Надто великий розмір вибірки зробить еволюцію повільною (на час роботи програми це не впливає, бо за весь великий цикл оновлюється 1000 розв’язків незалежно від розміру, але саме через це вся популяція оновиться меншу кількість разів, тому високе середнє значення буде досягнуто пізніше).

Побачимо, чи сповільнення відчутне, здійснивши по 20 еволюцій найменшого та найбільшого розмірів популяції з однаковими іншими параметрами (рис. 3.5-3.7):

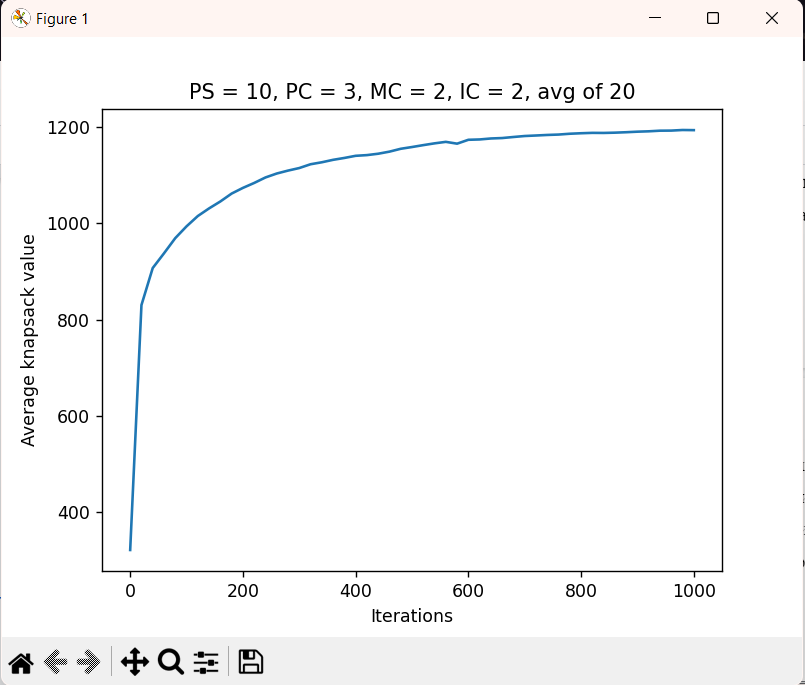


Рисунок 3.5 – Зростання вартості при розмірі вибірки в 10 розв’язків

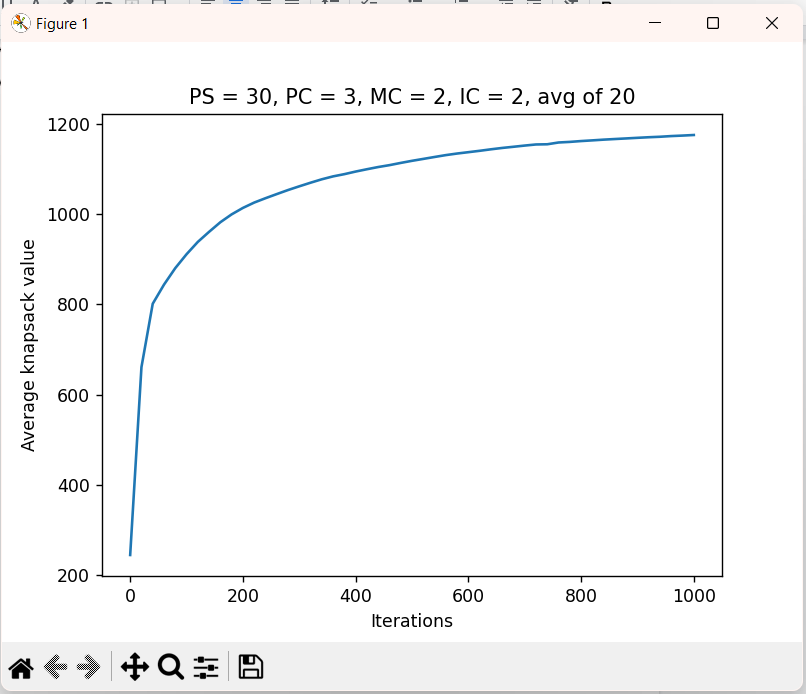


Рисунок 3.6 – Зростання вартості при розмірі вибірки в 30 розв’язків

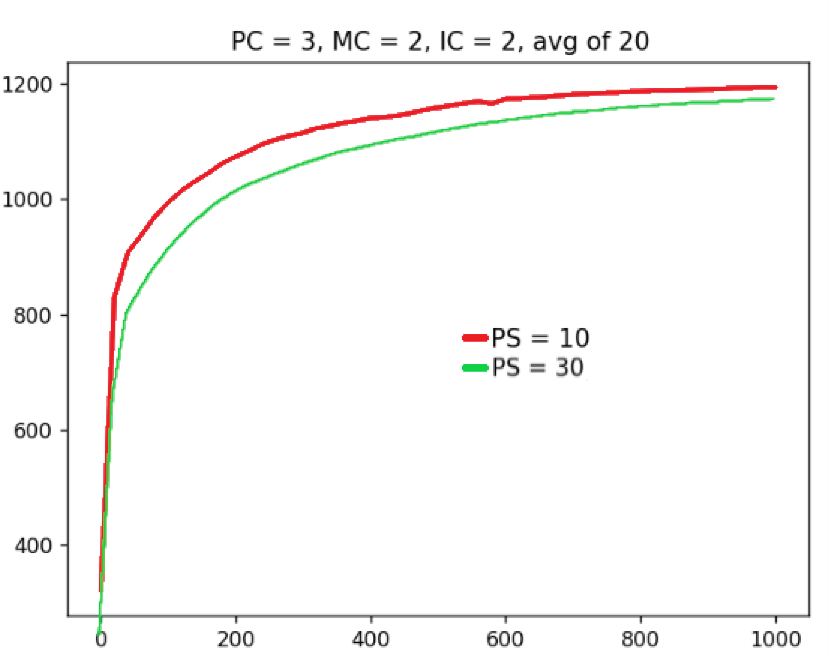


Рисунок 3.7 – Зіставивши два графіки, можемо побачити, що сповільнення є, але воно не є значним.

Тепер можна перевірити, за якого розміру популяції починається сповільнення, щоб обрати оптимальний розмір, за якого еволюція відбувається стабільно та швидко. Також, можна перевірити, чи є сповільнення закономірним, чи результат на рисунку 3.7 був випадковим (рис. 3.8):

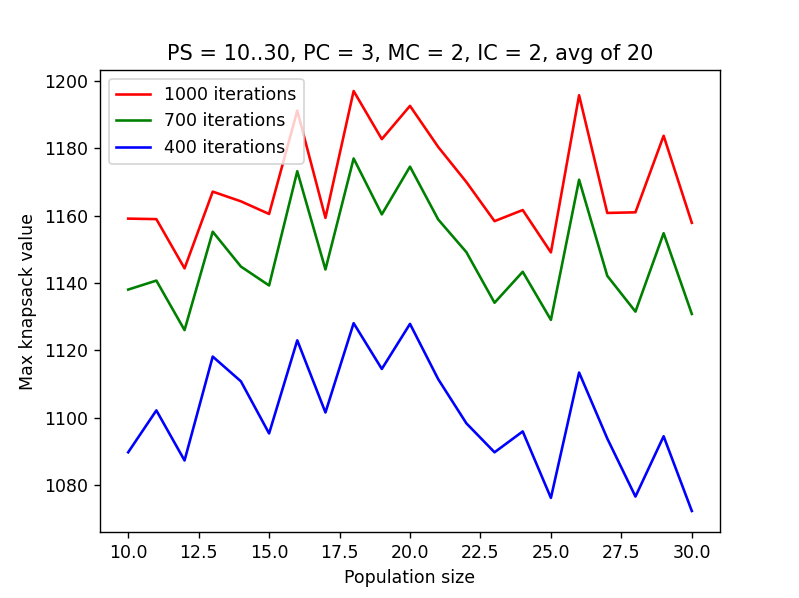


Рисунок 3.8 – Графік залежності максимальної вартості рюкзака від кількості виглядає майже випадковим

Інформація на графіку свідчить про те, що сповільнення, помічене на рис. 3.7, могло бути спричинено різною вартістю доступних предметів у наборі. Ця різна вартість не сильно впливає на результати (графік 3.8 сильно розтягнутий за віссю Y та ці відхилення можна назвати статистичною похибкою), але ми можемо побачити, як на лівій стороні рис. 3.8 відстані між червоною, зеленою та синьою ламаними ледь помітно збільшуються при збільшенні розміру популяції. Виділимо це на рис. 3.9-3.10:

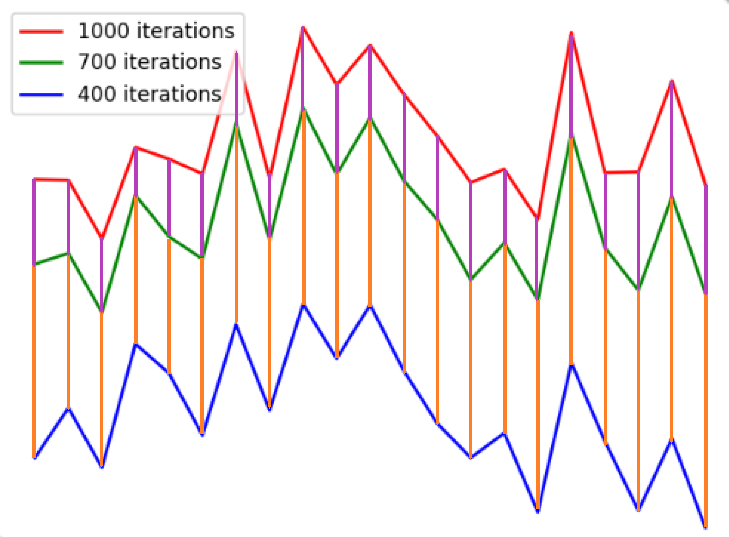


Рисунок 3.9 – Відстані між ламаними виділені

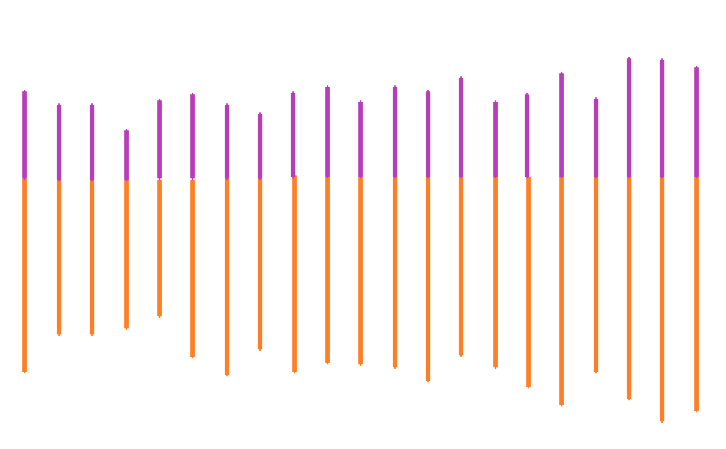
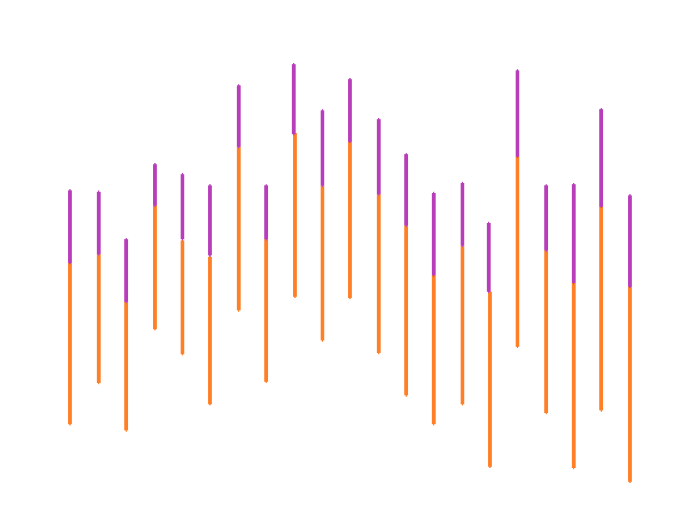


Рис. 3.10 – Ці ж відстані без ламаних та вирівняні за точками, що були на зеленій ламаній

Можна зробити висновок, що сповільненням зростання можна знехтувати, бо воно стає відчутним на розмірах популяції ще більших, ніж 30. Зафіксуємо PopulationSize = 30.

### Локальні покращення (improveCount)

В теорії, так як improveCount є єдиним параметром, що впливає не на випадкові процеси, а на ті, що гарантовано покращують вміст рюкзака, збільшення цього параметру значно зменшить кількість ітерацій, потрібних для “виходу на плато”. При цьому локальні покращення незначно збільшать час роботи програми. Також слід перевірити, чи не нашкодять локальні покращення, кожного разу вкладаючи до рюкзака важкі та дешеві предмети. Перевіримо теорію результатами випробування, що показані на рис. 3.11:

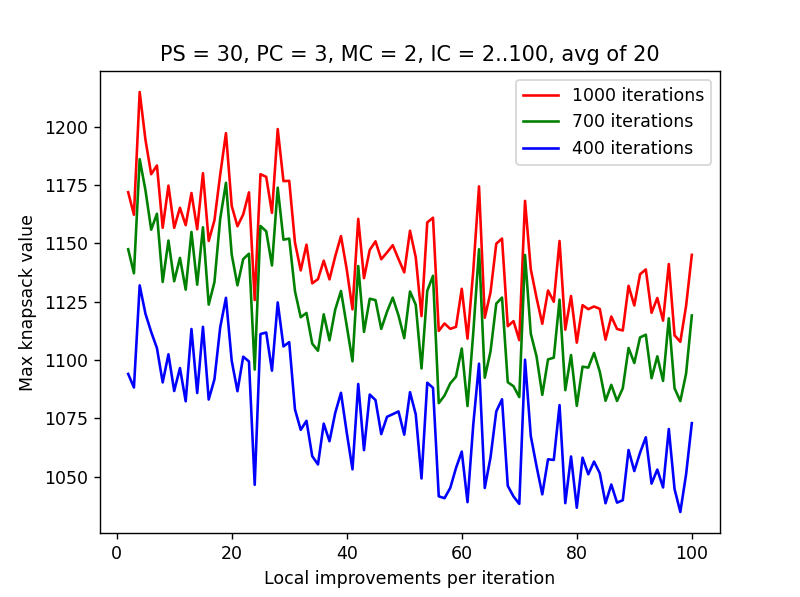


Рисунок 3.11 – Легко побачити, що вкладання важких та дешевих предметів трохи шкодить вартості рюкзака та повністю скасовує переваги методу “якомога швидше заповнити рюкзак чим завгодно”

Не можна виключати, що такий результат отримано тому, що відбувається мало мутацій та кросинговерів, які б звільняли місце в рюкзаку. Поки залишимо значення improveCount = 2.

### Мутації (mutationsCount)

Мутації роблять еволюцію більш хаотичною, але дозволяють вибити її з глухого кута, який можна спостерігати при великих значеннях improveCount. Результати зміни параметру mutationsCount при improveCount показано на рис. 3.12:

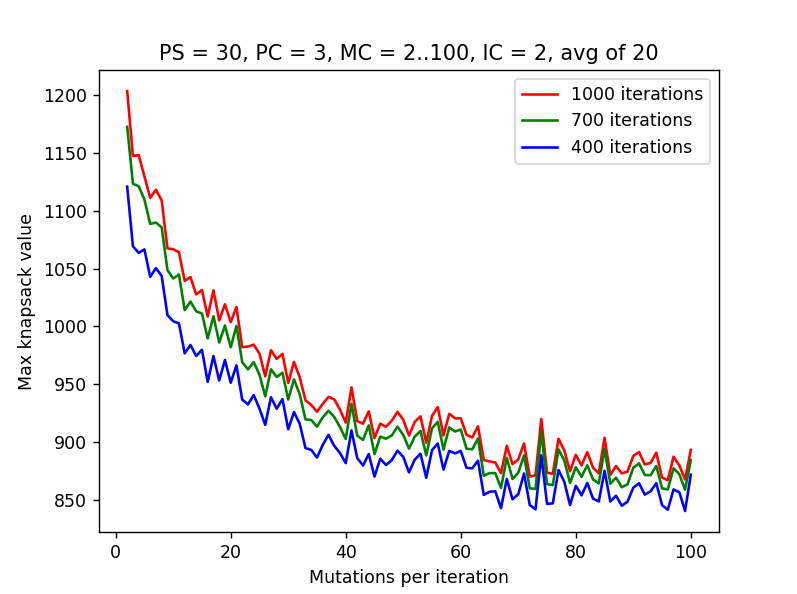


Рисунок 3.12 – Велика кількість мутацій, без втручання інших чинників, псує вдалі розв’язки

### Точки кросинговеру (pointsCount)

Перед тим, як переходити до балансування кількох показників одразу, слід отримати аналогічний графік для різної кількості кросинговерів (рис. 3.13):

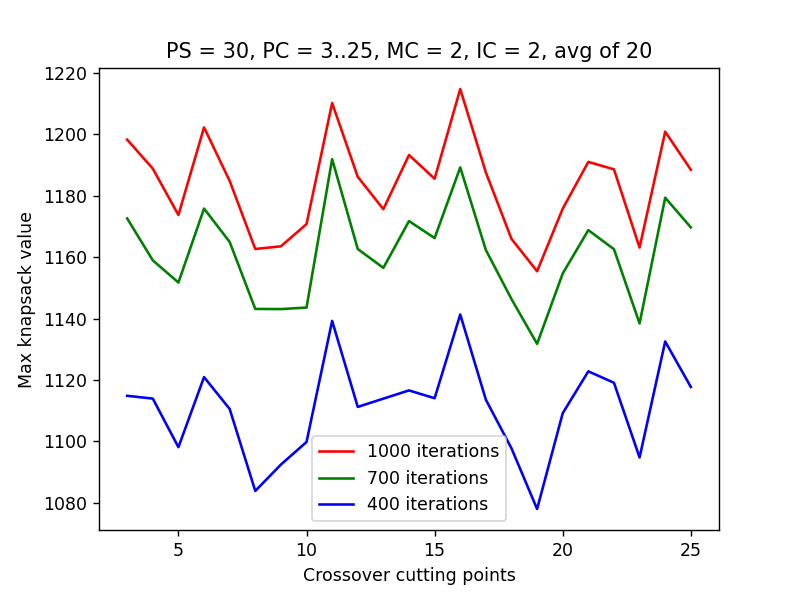


Рисунок 3.12 – За таких інших параметрів, між кількістю точок кросинговеру та вартістю рюкзака кореляції немає.

### Балансування значень pointsCount, mutationsCount та improveCount

Випробування проводитимуться за схемами на рис. 3.13-3.15:

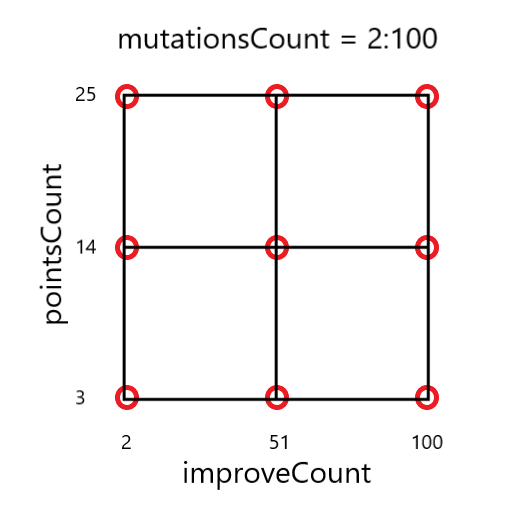


Рисунок 3.13 – Для кожної точки на графіку буде проведено випробування зі значеннями mutationsCount на проміжку від 2 до 100

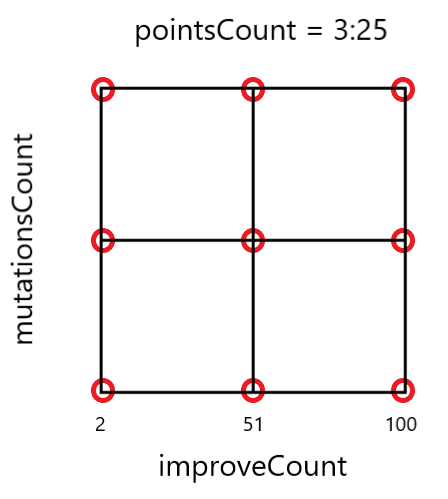


Рисунок 3.14 – Для кожної точки на графіку буде проведено випробування зі значеннями pointsCount на проміжку від 2 до 100

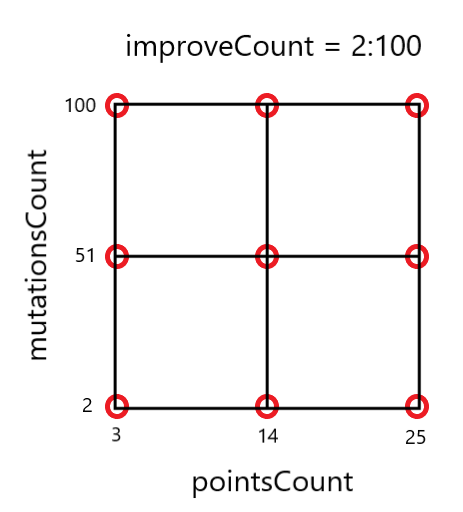
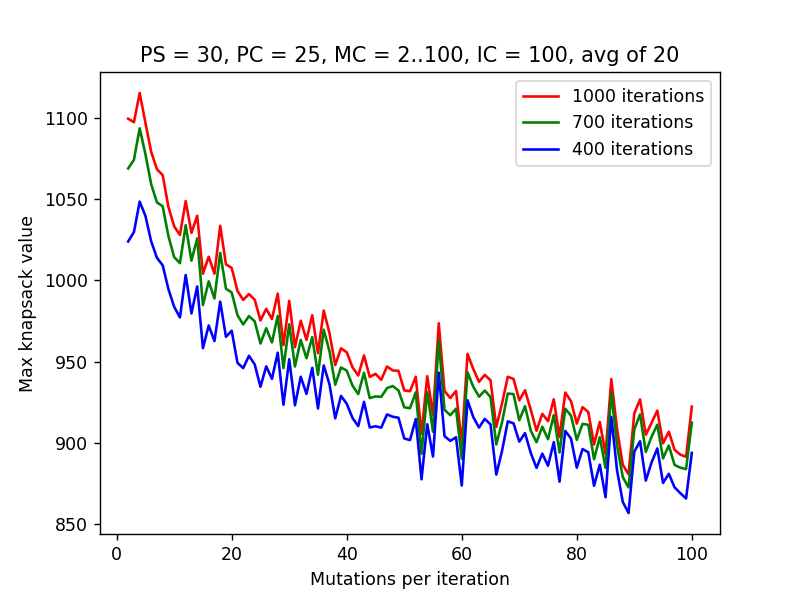
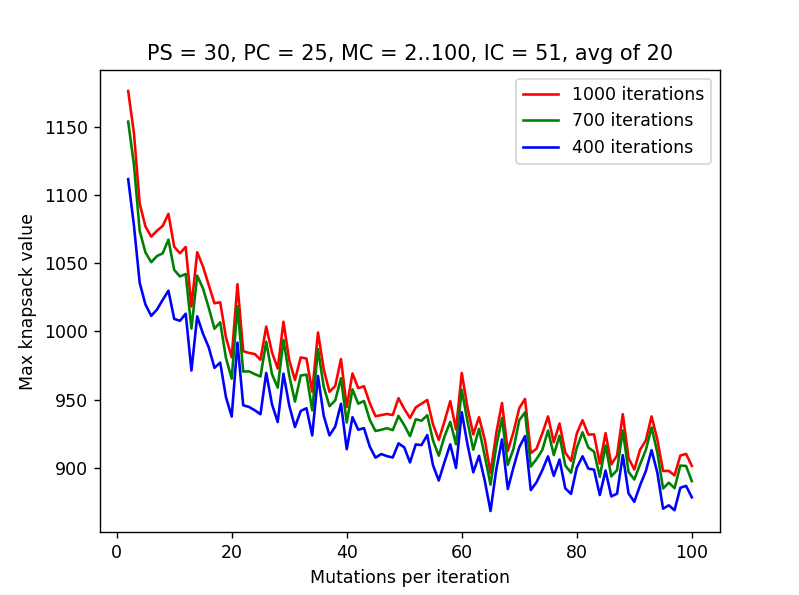
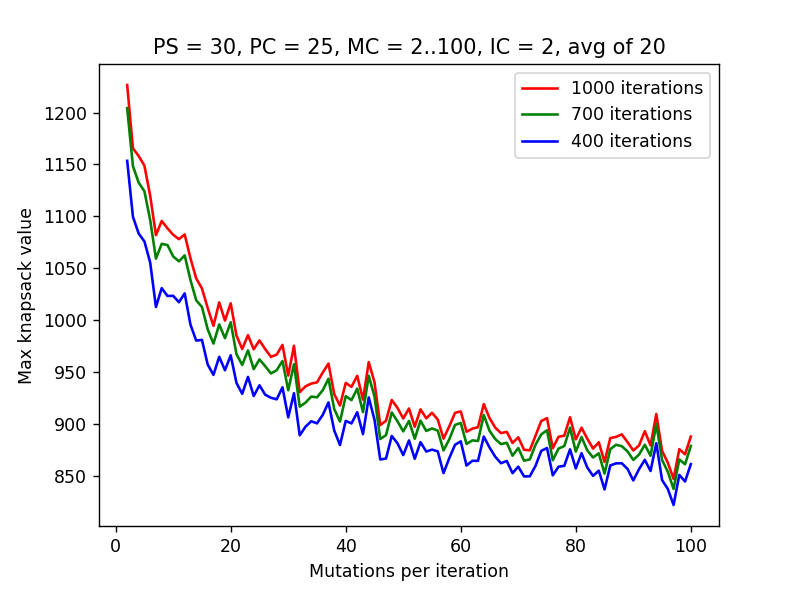
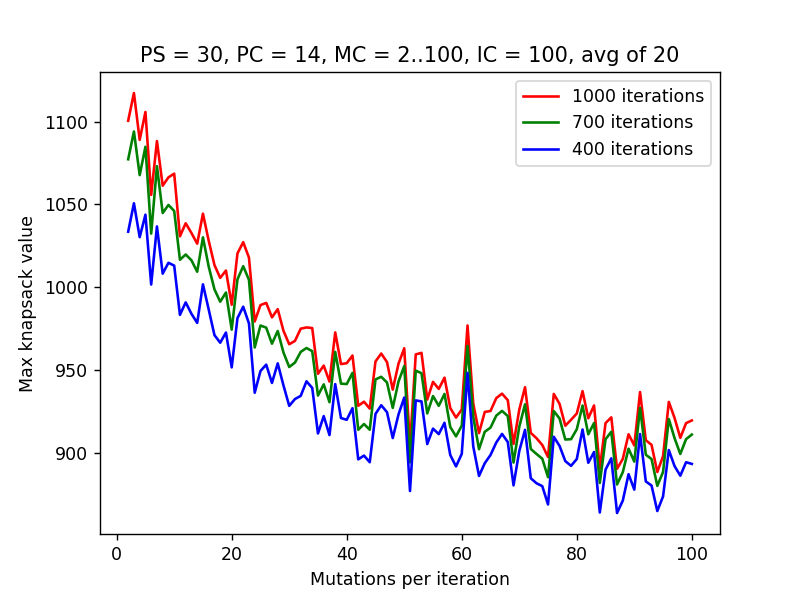
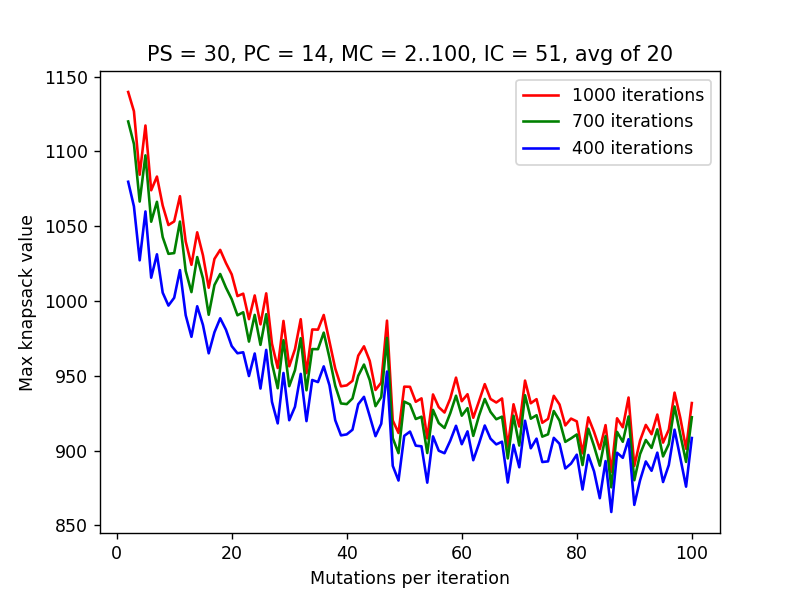
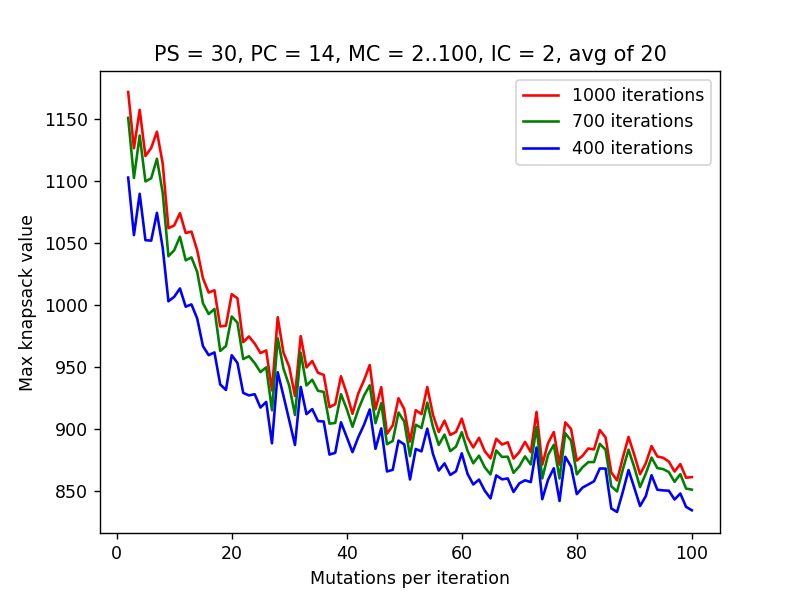
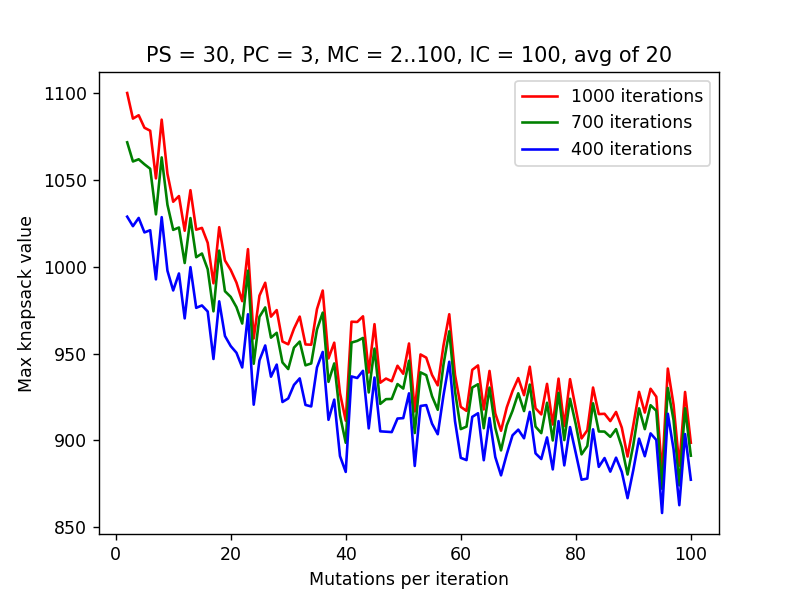
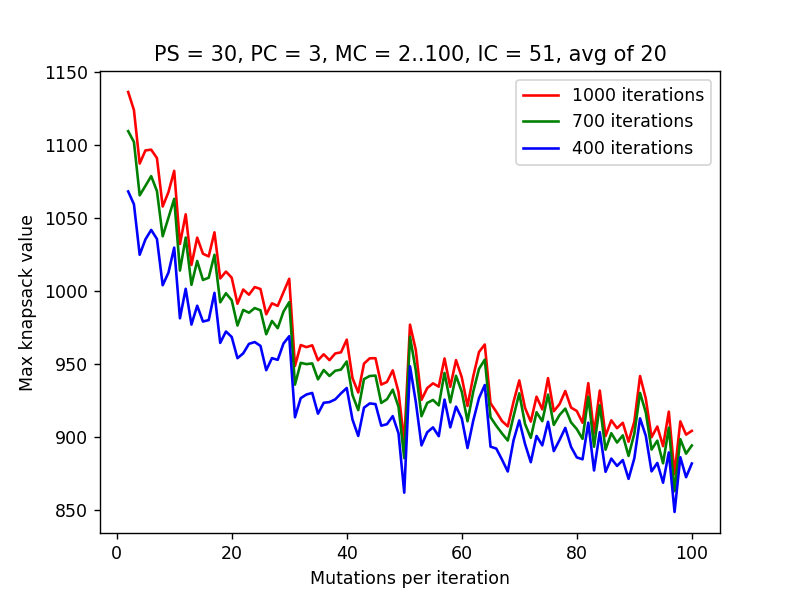
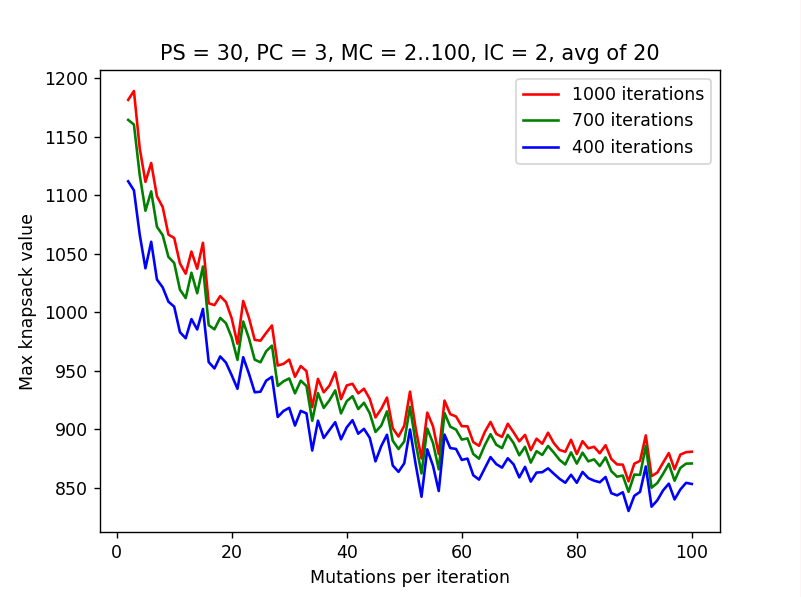


Рисунок 3.15 – Для кожної точки на графіку буде проведено випробування зі значеннями improveCount на проміжку від 2 до 100

Проведемо випробування (рис. 3.16-3.24):



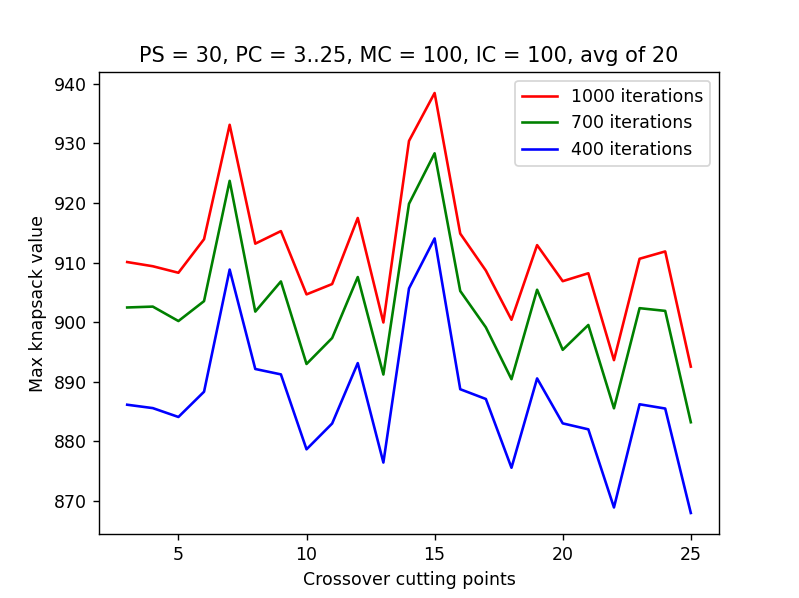
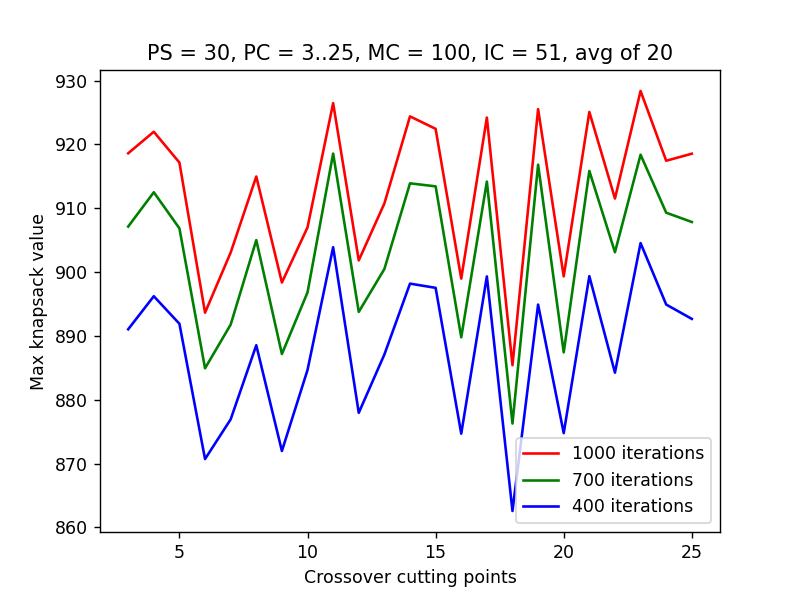
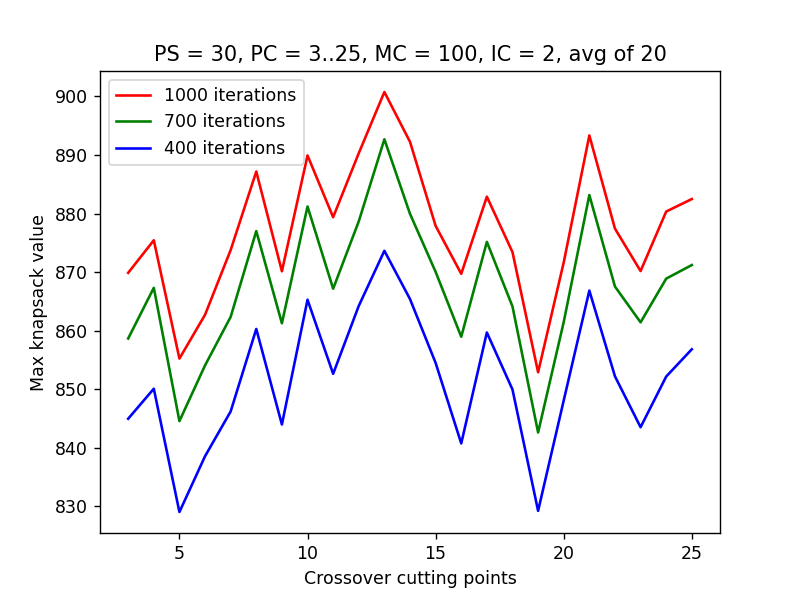


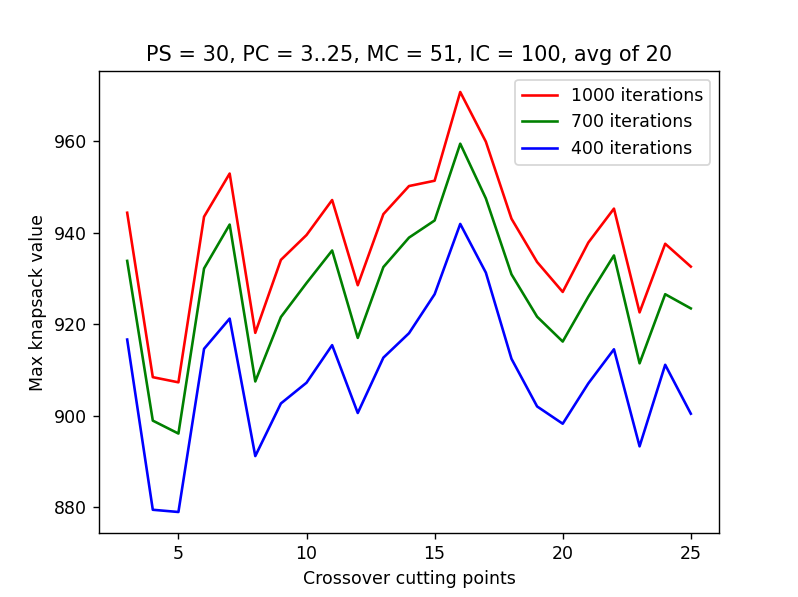
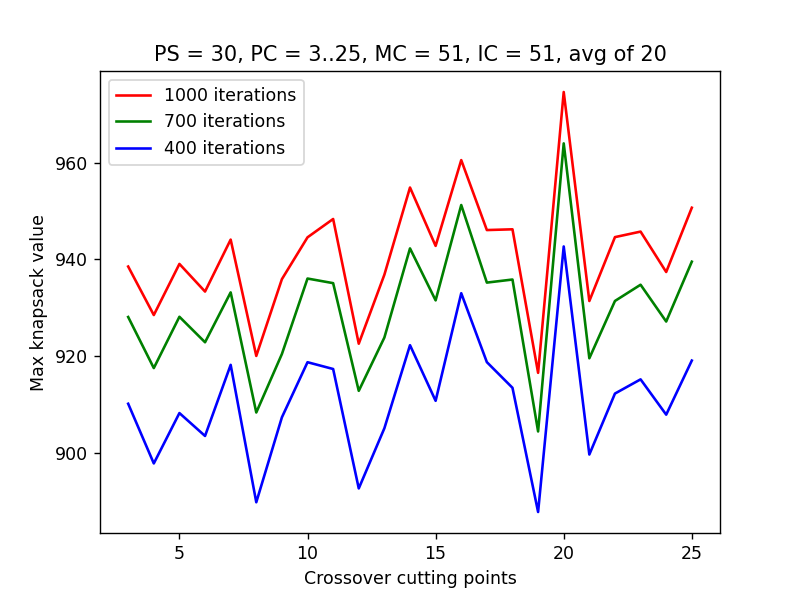
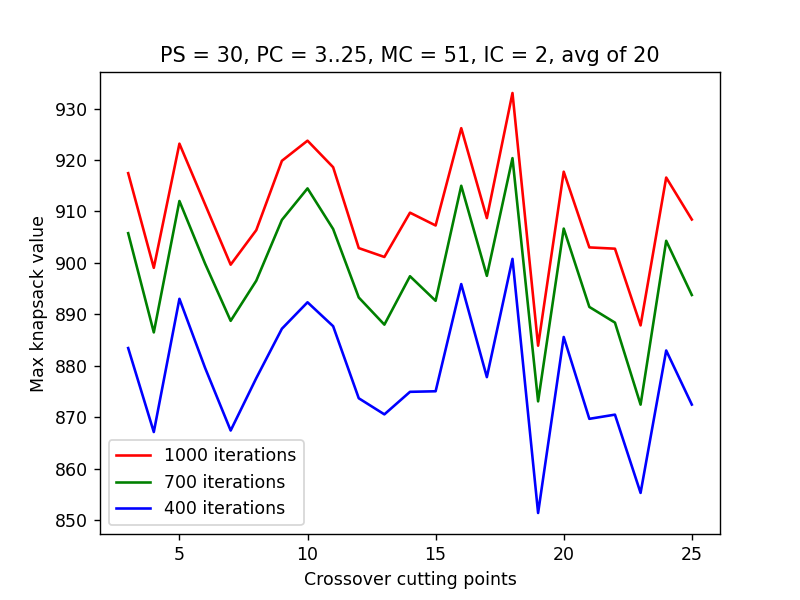


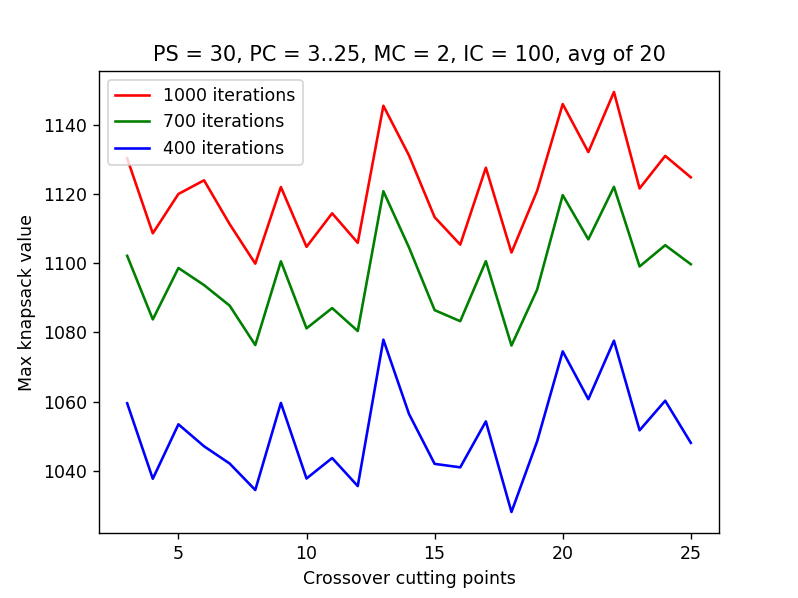
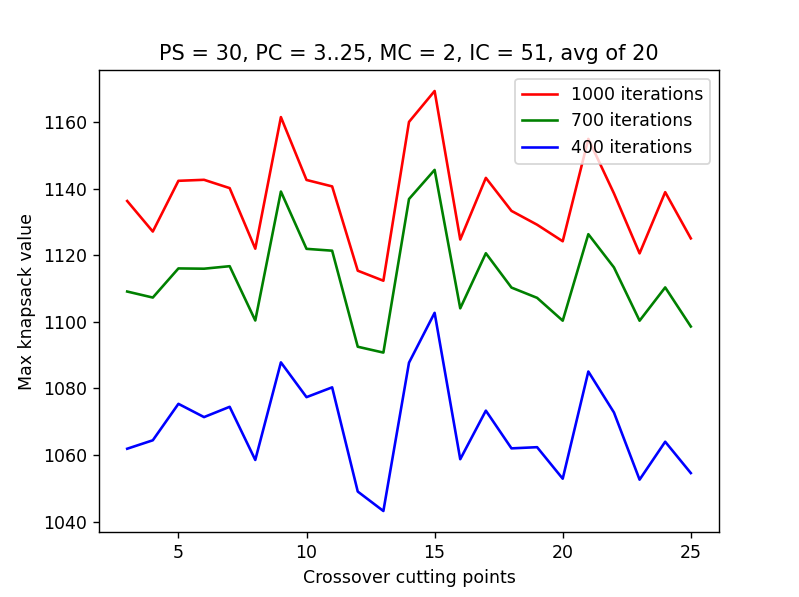
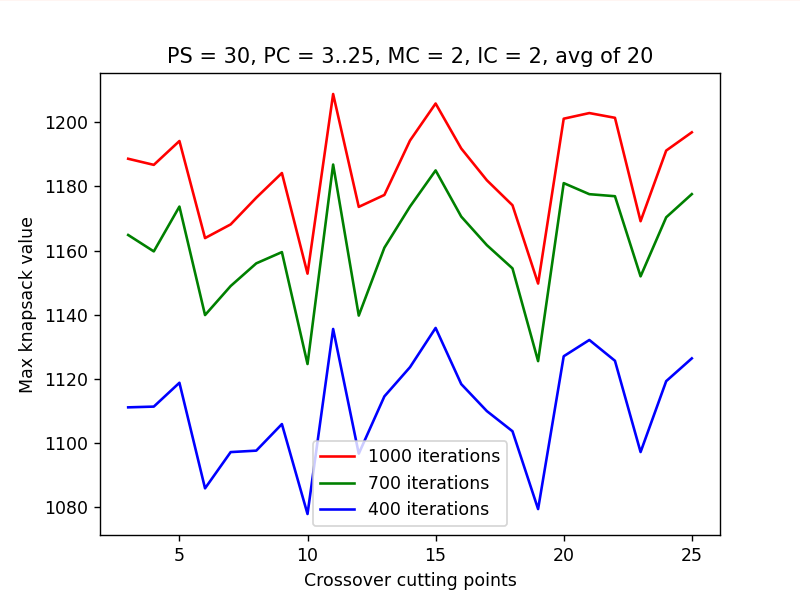
Рисунки 3.16-3.24 – Випробування, проведені за схемою на рис. 3.13

Результати випробувань можуть показувати один із двох можливих висновків: або найкраще значення параметра mutationsCount дорівнює 2, або існує така комбінація параметрів pointsCount та improveCount, за якої є позитивна кореляція між mutationsCount та максимумом середньої вартості рюкзака, але ця комбінація знаходиться далеко від червоних точок на рис. 3.13

Проведемо наступну серію випробувань (рис. 3.25-3.33):



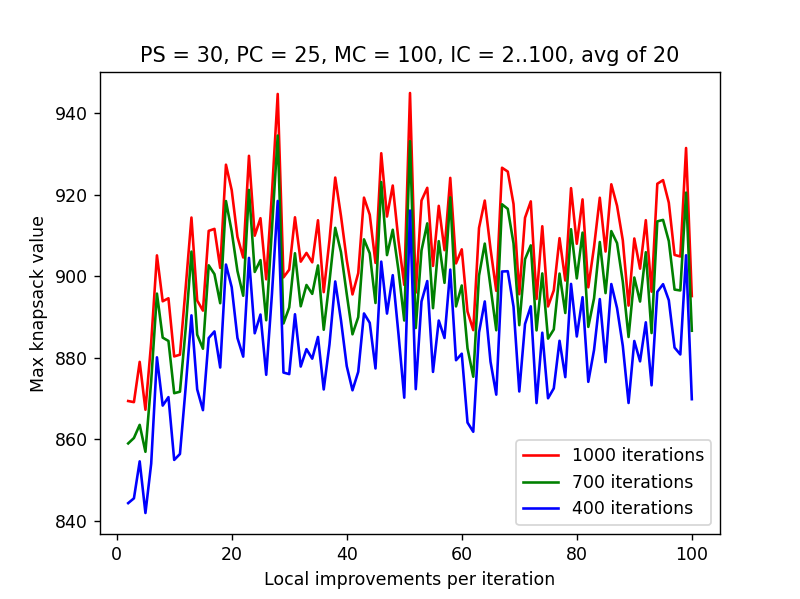
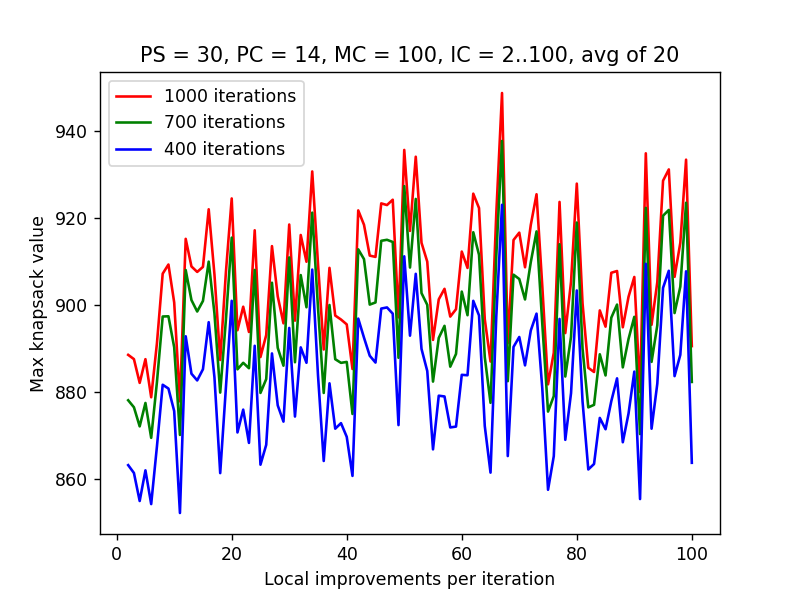
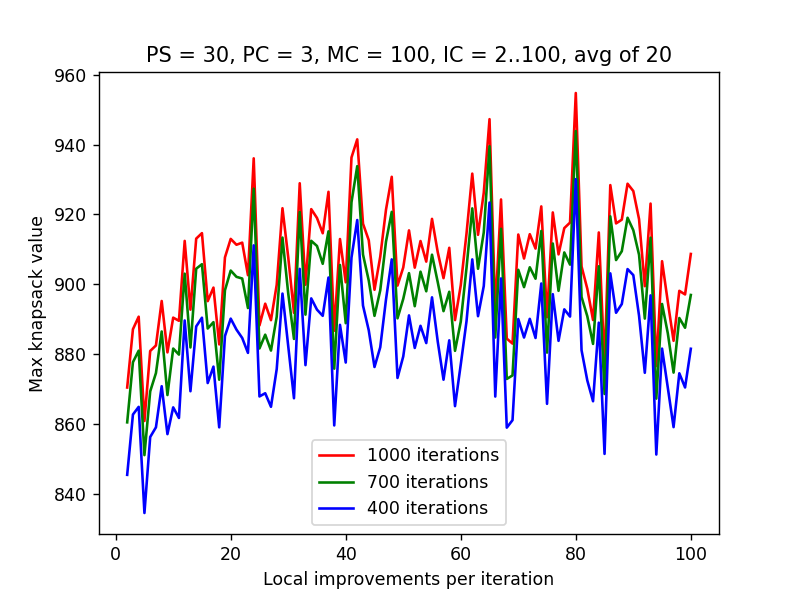


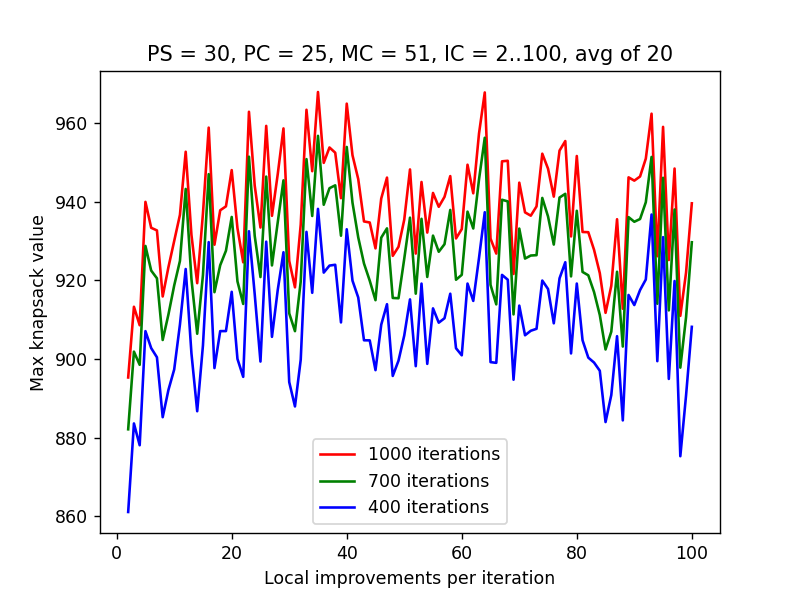
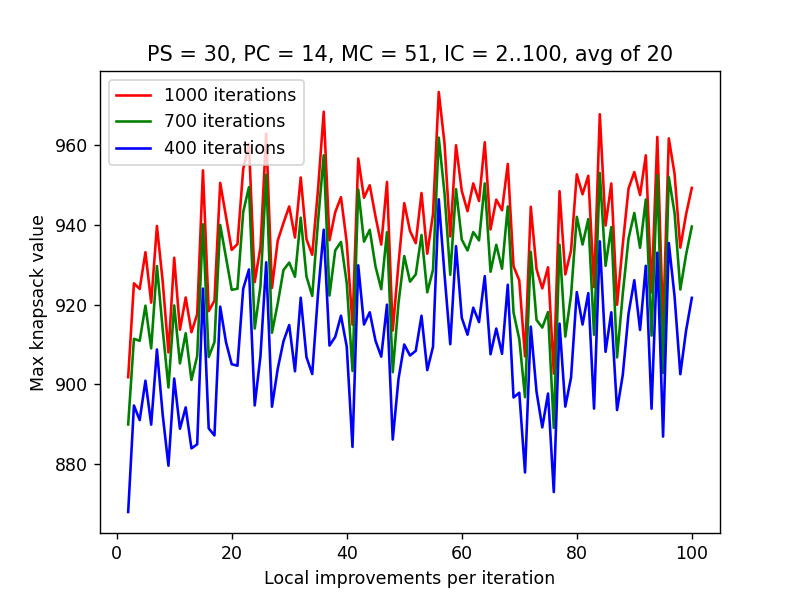
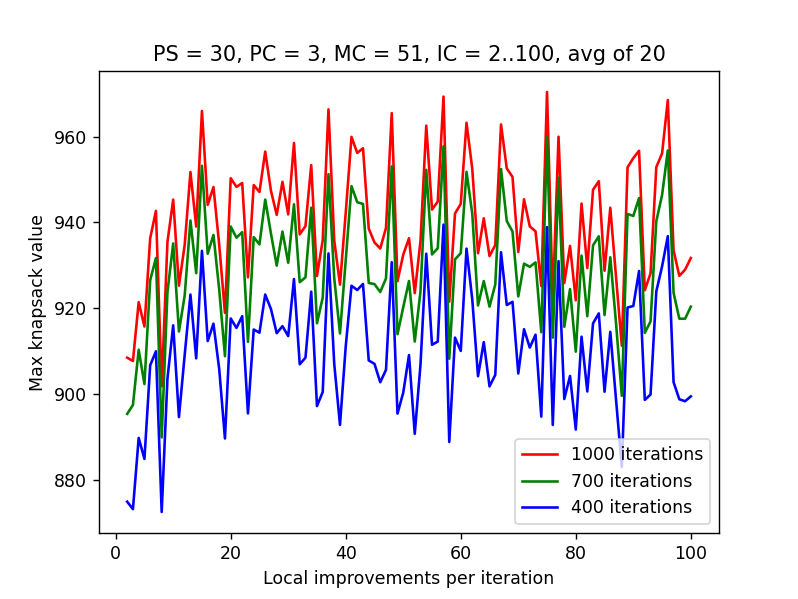


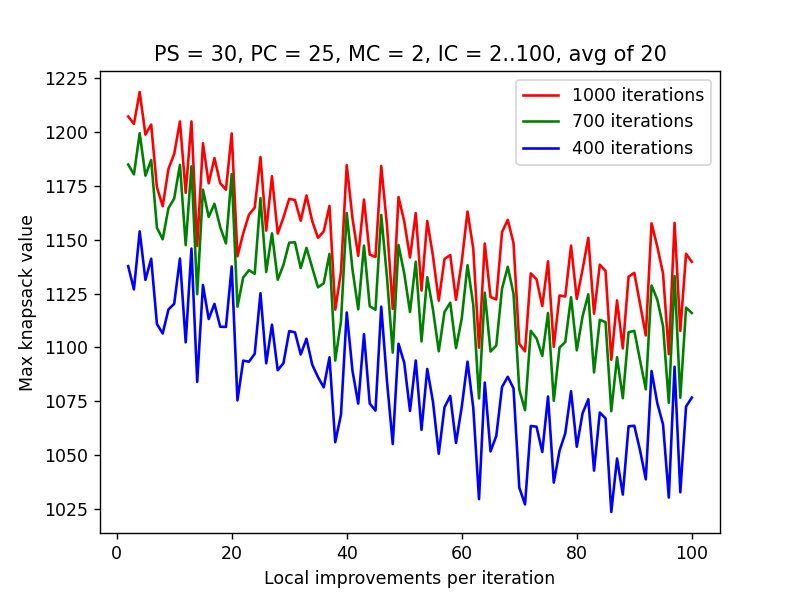
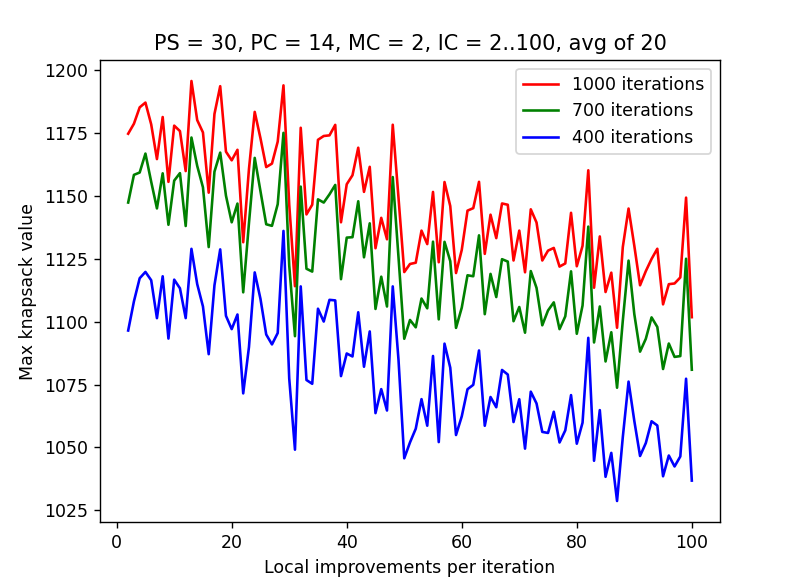
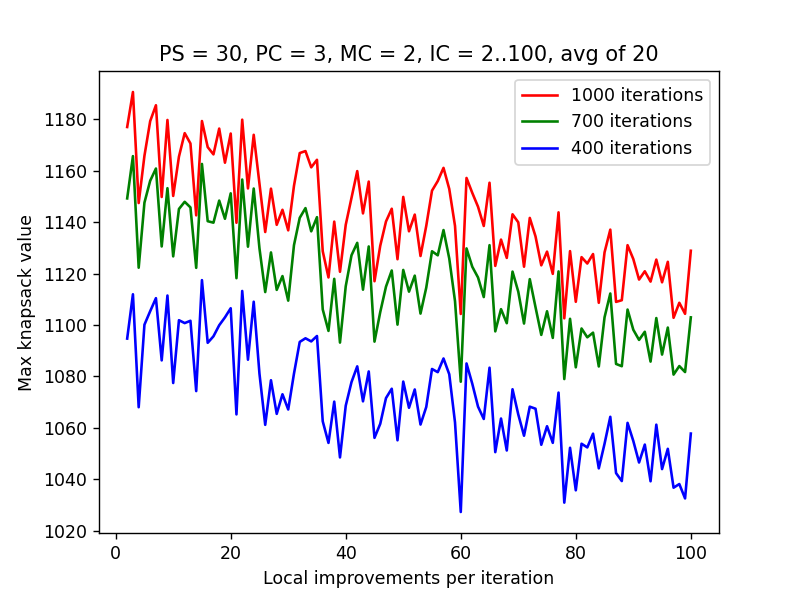
Рисунки 3.25-3.33 – Випробування, проведені за схемою на рис. 3.14

Результати випробувань показують на відсутність кореляції між значенням pointsCount та максимумом середньої вартості рюкзака, а також підтверджують, що є негативна кореляція між кількістю мутацій та вартістю рюкзака. Значення можна встановити: pointsCount = 3.

На рисунках 3.34-3.42 показано наступну серію випробувань:







Рисунки 3.34-3.42 – Випробування, проведені за схемою на рис. 3.15

На графіках видно, що їх форма залежить від кількості мутацій, яка сильно знижує вартість рюкзака, тому, в результаті всіх випробувань, можна сказати, що оптимальним (або близьким до оптимального) набором параметрів для даної симуляції є: populationSize = 30, pointsCount = 3, mutationsSize = 2, improveCount = 2.

Висновок

В рамках даної лабораторної роботи було розглянуто роботу генетичного алгоритму на прикладі задачі про рюкзак. Досліджено переваги та недоліки алгоритму.

Розроблено програмне забезпечення для роботи генетичного алгоритму з різними параметрами: розмір вибірки, кількість точок кросинговеру, кількість мутацій, а також кількість локальних покращень.

Проведено аналіз роботи алгоритму за різних параметрів та знайдено набір параметрів, за якого середнє значення вибірки отриманих розв’язків задачі є близьким до максимального.

Критерії оцінювання

При здачі лабораторної роботи до 11.12.2022 включно максимальний бал дорівнює – 5. Після 11.12.2022 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

* покроковий алгоритм – 15%;
* програмна реалізація алгоритму – 50%;
* тестування алгоритму– 30%;
* висновок – 5%.