**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**”

Варіант 1

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

ІП-з21 Стихун М.В.

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.Н.*

Київ 2023

Зміст

1. Мета лабораторної роботи 3
2. Завдання 4
3. Виконання 12
   1. Покроковий алгоритм 12
   2. Програмна реалізація алгоритму 12
      1. Вихідний код 12
      2. Приклади роботи 37

Критерії оцінювання 39

# 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.  Рисунок 1  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

## Покроковий алгоритм

…

## Програмна реалізація алгоритму

### Вихідний код

package main

import (

"flag"

"github.com/Nav1Cr0ss/algorithms/internal/domain/lab\_1"

"github.com/Nav1Cr0ss/algorithms/pkg/sort"

)

func main() {

var inputFileName string

var outputFileName string

var useExternalSort bool

flag.StringVar(&inputFileName, "input", "input.txt", "Input file name")

flag.StringVar(&outputFileName, "output", "output.txt", "Output file name")

flag.BoolVar(&useExternalSort, "external", false, "Use external sort (default is in-memory sort)")

flag.Parse()

var sorter lab\_1.SortProvider

switch useExternalSort {

case true:

sorter = sort.NewExternalSort()

default:

sorter = sort.NewMemorySort()

}

lab\_1.SortArrayOfInt(inputFileName, outputFileName, sorter)

}

package main

import (

"bufio"

"github.com/Nav1Cr0ss/algorithms/internal/domain/lab\_1"

"github.com/Nav1Cr0ss/algorithms/pkg/fs"

"github.com/Nav1Cr0ss/algorithms/pkg/sort"

"os"

"strconv"

"testing"

)

func TestSorting(t \*testing.T) {

tests := []struct {

name string

sortProvider lab\_1.SortProvider

inputFileName string

outputFileName string

totalNum int

}{

{

name: "MemorySort",

sortProvider: sort.NewMemorySort(),

inputFileName: "test\_memory\_input.txt",

outputFileName: "test\_memory\_output.txt",

totalNum: 1000000,

},

{

name: "ExternalSort",

sortProvider: sort.NewExternalSort(),

inputFileName: "test\_external\_input.txt",

outputFileName: "test\_external\_output.txt",

totalNum: 100000000,

},

}

for \_, tt := range tests {

t.Run(tt.name, func(t \*testing.T) {

fileSystem := fs.NewFS()

if err := generateInputFile(fileSystem, tt.inputFileName, tt.totalNum); err != nil {

panic(err)

}

defer func() {

if err := deleteOutputFile(fileSystem, []string{tt.inputFileName, tt.outputFileName}); err != nil {

panic(err)

}

}()

lab\_1.SortArrayOfInt(tt.inputFileName, tt.outputFileName, tt.sortProvider)

outputFile, err := os.Open(tt.outputFileName)

if err != nil {

t.Fatalf("Failed to open %s: %v", tt.outputFileName, err)

}

defer outputFile.Close()

scanner := bufio.NewScanner(outputFile)

numbers := make([]int, 100)

for i := 0; i < 100; i++ {

if !scanner.Scan() {

t.Fatalf("Failed to read 100 numbers from %s", tt.outputFileName)

}

num, err := strconv.Atoi(scanner.Text())

if err != nil {

t.Fatalf("Failed to parse number from %s: %v", tt.outputFileName, err)

}

numbers[i] = num

}

if !isSorted(numbers) {

t.Fatalf("The numbers in %s are not sorted", tt.outputFileName)

}

})

}

}

func isSorted(numbers []int) bool {

for i := 1; i < len(numbers); i++ {

if numbers[i-1] > numbers[i] {

return false

}

}

return true

}

func generateInputFile(fs \*fs.FS, fileName string, totalNum int) error {

err := fs.CreateFileWithArrOfInt(fileName, totalNum)

if err != nil {

return err

}

return nil

}

func deleteOutputFile(fs \*fs.FS, fileNames []string) error {

err := fs.DeleteFiles(fileNames)

if err != nil {

return err

}

return nil

}

package lab\_1

import "bufio"

type Item struct {

value int

index int

scanner \*bufio.Scanner

}

type PriorityQueue []\*Item

func (pq PriorityQueue) Len() int { return len(pq) }

func (pq PriorityQueue) Less(i, j int) bool {

return pq[i].value < pq[j].value

}

func (pq PriorityQueue) Swap(i, j int) {

pq[i], pq[j] = pq[j], pq[i]

pq[i].index = i

pq[j].index = j

}

func (pq \*PriorityQueue) Push(x interface{}) {

item := x.(\*Item)

item.index = len(\*pq)

\*pq = append(\*pq, item)

}

func (pq \*PriorityQueue) Pop() interface{} {

old := \*pq

n := len(old)

item := old[n-1]

item.index = -1

\*pq = old[0 : n-1]

return item

}

package lab\_1

import (

"fmt"

"github.com/Nav1Cr0ss/algorithms/pkg/utilz"

"time"

)

type SortProvider interface {

Sort(inputFile, outputFile string) error

}

func SortArrayOfInt(inputFile, outputFile string, sp SortProvider) {

start := time.Now()

err := sp.Sort(inputFile, outputFile)

if err != nil {

fmt.Println("Error on sorting file")

}

utilz.MeasureTime(start, "Sorting array")

}

package fs

import (

"bufio"

"fmt"

"log"

"math/rand"

"os"

"strconv"

"sync"

)

type FS struct{}

func NewFS() \*FS {

return &FS{}

}

func (f \*FS) DeleteFile(fileName string) error {

err := os.Remove(fileName)

if err != nil {

return err

}

return nil

}

func (f \*FS) ParseIntArrayFromFile(filename string) ([]int, error) {

file, err := os.Open(filename)

if err != nil {

return nil, err

}

defer file.Close()

scanner := bufio.NewScanner(file)

scanner.Buffer(make([]byte, 64\*1024), 1024\*1024)

var nums []int

for scanner.Scan() {

line := scanner.Text()

num, err := strconv.Atoi(line)

if err != nil {

fmt.Println("Error parsing number:", err)

continue

}

nums = append(nums, num)

}

if err := scanner.Err(); err != nil {

fmt.Println("Error scanning file:", err)

return nums, err

}

return nums, nil

}

func (f \*FS) WriteIntArrayToFile(filename string, arr []int) error {

outputFile, err := os.Create(filename)

if err != nil {

fmt.Println("Error creating file:", err)

return err

}

defer outputFile.Close()

writer := bufio.NewWriter(outputFile)

defer writer.Flush()

for \_, num := range arr {

\_, err := writer.WriteString(strconv.Itoa(num) + "\n")

if err != nil {

fmt.Println("Error writing to file:", err)

return err

}

}

return nil

}

func (f \*FS) WriteChunk(filename string, data []int) error {

file, err := os.Create(filename)

if err != nil {

return err

}

defer file.Close()

writer := bufio.NewWriter(file)

defer writer.Flush()

for \_, num := range data {

\_, err := writer.WriteString(strconv.Itoa(num) + "\n")

if err != nil {

return err

}

}

return nil

}

func (f \*FS) MergeChunks(chunkFiles []string, outputFile string) error {

output, err := os.Create(outputFile)

if err != nil {

return err

}

defer output.Close()

chunkData := make(chan string)

var wg sync.WaitGroup

for \_, chunkFile := range chunkFiles {

wg.Add(1)

go func(file string) {

defer wg.Done()

chunkReader, err := os.Open(file)

if err != nil {

log.Println(err)

return

}

defer chunkReader.Close()

scanner := bufio.NewScanner(chunkReader)

for scanner.Scan() {

chunkData <- scanner.Text()

}

if scanner.Err() != nil {

log.Println(scanner.Err())

}

}(chunkFile)

}

go func() {

writer := bufio.NewWriter(output)

defer writer.Flush()

for data := range chunkData {

fmt.Fprintln(writer, data)

}

}()

wg.Wait()

close(chunkData)

return nil

}

func (f \*FS) DeleteFiles(fileNames []string) error {

for \_, name := range fileNames {

err := f.DeleteFile(name)

if err != nil {

return err

}

}

return nil

}

func (f \*FS) CreateFileWithArrOfInt(filename string, totalNumbers int) error {

outputFile, err := os.Create(filename)

if err != nil {

fmt.Println("Error creating file:", err)

return err

}

defer func(outputFile \*os.File) {

err = outputFile.Close()

if err != nil {

panic("panic on closing file")

}

}(outputFile)

writer := bufio.NewWriter(outputFile)

defer func(writer \*bufio.Writer) {

err = writer.Flush()

if err != nil {

panic("panic on flushing file")

}

}(writer)

for i := 0; i < totalNumbers; i++ {

randomNum := rand.Intn(1000000)

\_, err := writer.WriteString(strconv.Itoa(randomNum) + "\n")

if err != nil {

fmt.Println("Error writing to file:", err)

return err

}

}

return nil

}

package main

import (

"flag"

"github.com/Nav1Cr0ss/algorithms/pkg/fs"

//fs2 "github.com/Nav1Cr0ss/algorithms/pkg/fs"

"github.com/Nav1Cr0ss/algorithms/pkg/utilz"

"time"

)

func main() {

var fileName string

flag.StringVar(&fileName, "file\_name", "input.txt", "file name")

files := fs.NewFS()

start := time.Now()

err := files.DeleteFile(fileName)

if err != nil {

return

}

utilz.MeasureTime(start, "Deleting")

}

package main

import (

"flag"

"github.com/Nav1Cr0ss/algorithms/pkg/fs"

//fs2 "github.com/Nav1Cr0ss/algorithms/pkg/fs"

"github.com/Nav1Cr0ss/algorithms/pkg/utilz"

"time"

)

func main() {

var fileName string

var numsToGenerate int

flag.StringVar(&fileName, "file\_name", "input.txt", "file name")

flag.IntVar(&numsToGenerate, "output", 100000, "Nums to generate")

files := fs.NewFS()

start := time.Now()

err := files.CreateFileWithArrOfInt(fileName, numsToGenerate)

if err != nil {

return

}

utilz.MeasureTime(start, "Generating array")

}

package sort

import (

"bufio"

"fmt"

"github.com/Nav1Cr0ss/algorithms/pkg/fs"

"os"

"sort"

"strconv"

)

const *maxMemory* = int64(100 \* 1024 \* 1024)

type ExternalSort struct {

fs \*fs.FS

}

func NewExternalSort() \*ExternalSort {

return &ExternalSort{

fs: fs.NewFS(),

}

}

func (s \*ExternalSort) Sort(inputFile, outputFile string) error {

chunkFiles, err := s.createSortedChunks(inputFile, *maxMemory*)

if err != nil {

return err

}

defer func(files []string) {

err = s.fs.DeleteFiles(files)

if err != nil {

}

}(chunkFiles)

err = s.fs.MergeChunks(chunkFiles, outputFile)

if err != nil {

return err

}

return nil

}

func (s \*ExternalSort) createSortedChunks(inputFile string, maxMemory int64) ([]string, error) {

file, err := os.Open(inputFile)

if err != nil {

return nil, err

}

defer file.Close()

chunkFiles := []string{}

scanner := bufio.NewScanner(file)

buffer := []int{}

memUsed := int64(0)

for scanner.Scan() {

line := scanner.Text()

num, err := strconv.Atoi(line)

if err != nil {

return nil, err

}

buffer = append(buffer, num)

memUsed += int64(len(line) + 1)

if memUsed >= maxMemory {

sort.Ints(buffer)

chunkFile := fmt.Sprintf("chunk%d.txt", len(chunkFiles))

err := s.fs.WriteChunk(chunkFile, buffer)

if err != nil {

return nil, err

}

chunkFiles = append(chunkFiles, chunkFile)

buffer = []int{}

memUsed = 0

}

}

if len(buffer) > 0 {

sort.Ints(buffer)

chunkFile := fmt.Sprintf("chunk%d.txt", len(chunkFiles))

err := s.fs.WriteChunk(chunkFile, buffer)

if err != nil {

return nil, err

}

chunkFiles = append(chunkFiles, chunkFile)

}

return chunkFiles, nil

}

package sort

import "github.com/Nav1Cr0ss/algorithms/pkg/fs"

type MemorySort struct {

fs \*fs.FS

}

func NewMemorySort() \*MemorySort {

return &MemorySort{

fs: fs.NewFS(),

}

}

func (s \*MemorySort) merge(left, right []int) []int {

result := make([]int, 0, len(left)+len(right))

l, r := 0, 0

for l < len(left) && r < len(right) {

if left[l] < right[r] {

result = append(result, left[l])

l++

} else {

result = append(result, right[r])

r++

}

}

result = append(result, left[l:]...)

result = append(result, right[r:]...)

return result

}

func (s \*MemorySort) MergeSort(arr []int) []int {

if len(arr) <= 1 {

return arr

}

mid := len(arr) / 2

left := s.MergeSort(arr[:mid])

right := s.MergeSort(arr[mid:])

return s.merge(left, right)

}

func (s \*MemorySort) Sort(inputFile, outputFile string) error {

nums, err := s.fs.ParseIntArrayFromFile(inputFile)

if err != nil {

return err

}

sortedNums := s.MergeSort(nums)

err = s.fs.WriteIntArrayToFile(outputFile, sortedNums)

if err != nil {

return err

}

return nil

}

package generate

//go:generate go run pkg/scripts/del\_file/main.go

//go:generate go run pkg/scripts/gen\_rand\_num\_file/main.go

### Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

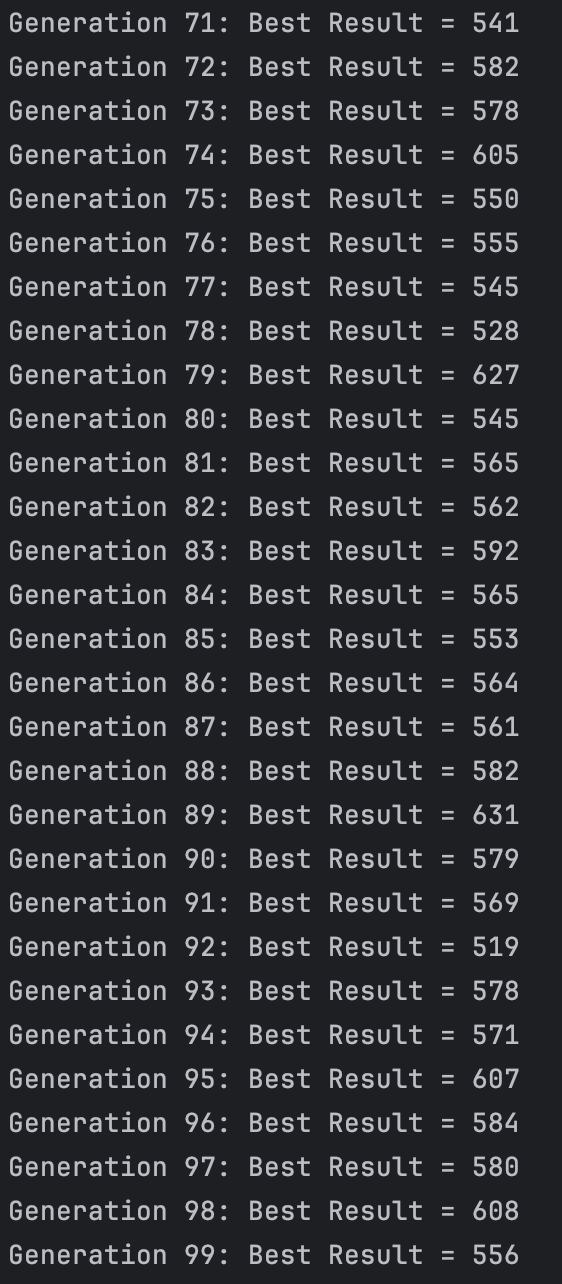
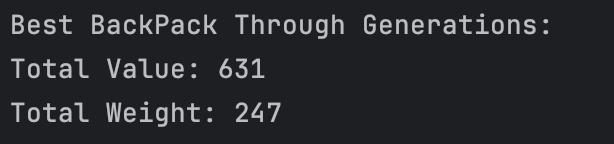
Рисунок 3.1 – Найкраще значення розбите по генераціям

Рисунок 3.2 –Накраще значення через усіх генерацій

Висновок

У ході виконання лабораторної роботи було досліджено та реалізовано два важливих алгоритми сортування: сортування в пам'яті та зовнішнього сортування.

Сортування в пам'яті є дієвим методом для сортування невеликих обсягів даних, коли весь набір даних поміщається у пам'ять комп'ютера.

Ми вивчили реалізацію алгоритму сортування злиттям **(Merge Sort)** та виконали практичну реалізацію цього алгоритму у мові програмування Go. Сортування в пам'яті є ефективним та швидким, але має обмеження на обсяг даних, які можна обробити.

Зовнішнє сортування **(External Sort)** є методом сортування для дуже великих обсягів даних, які не вміщуються у пам'ять. Ми дослідили алгоритм зовнішнього сортування, який вимагає поділу даних на менші частини (чанки), їх сортування та об'єднання в правильному порядку. За для досягнення максимальної швидкості, запис в банки проводив п паралельному форматі. Цей метод дозволяє сортувати навіть дуже великі набори даних, але вимагає більше обчислювальних та дискових ресурсів.

Критерії оцінювання

При здачі лабораторної роботи до 24.12.2023 включно максимальний бал дорівнює – 5. Після 24.12.2023 максимальний бал дорівнює – 4,5.

Критерії оцінювання у відсотках від максимального балу:

* покроковий алгоритм – 10%;
* програмна реалізація алгоритму – 45%;
* робота з гіт – 20%;
* тестування алгоритму– 20%;
* висновок – 5%.

+1 додатковий бал можна отримати за виконання та захист роботи до 17.12.2023