Knapsack

0.1

Generated by Doxygen 1.9.6

Chapter 1

Task:

Write a program that is designed to solve the knapsack problem using a genetic algorithm. Program should take input data from an input file that hols a set of items with their names, masses and values. Program should find a subset of items with maximal value and total weight that does not exceed capacity of the knapsack provided by user.
This program can be used from the command line by passing the following switches:
-i name.txt : input file with a set of items
-o name.txt : output file with the best solutions in all generations
-c value : knapsack capacity
-g value : number of generations
-n value : number of individuals in a generation

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Class Documentation

4.1 Item Struct Reference

Struct representing an item in knapsack problem.

```
#include <structures.h>
```

Public Attributes

- std::string name
- double weight
- int value

4.1.1 Detailed Description

Struct representing an item in knapsack problem.

Definition at line 19 of file structures.h.

4.1.2 Member Data Documentation

4.1.2.1 name

```
std::string Item::name
```

The name of the item

Definition at line 20 of file structures.h.

4.1.2.2 value

```
int Item::value
```

The value of the item

Definition at line 22 of file structures.h.

4.1.2.3 weight

```
double Item::weight
```

The weight of the item

Definition at line 21 of file structures.h.

The documentation for this struct was generated from the following file:

- structures.h

Chapter 5

File Documentation

5.1 functions.cpp File Reference

File containing necessary functions for program to work.

```
#include <iostream>
#include <fstream>
#include <random>
#include <algorithm>
#include <chrono>
#include <cmath>
#include <vector>
#include <string>
#include "structures.h"
```

Functions

- std::vector< Item > loadItems (const std::string &fileName)

  Loads a list of items from a file and returns them as a vector.
- double calculateFitness (const std::vector< int > &solution, const std::vector< Item > &items, double &weightLimit)

  Calculates the fitness of a solution using a list of items and a weight limit.
- void mutate (std::vector< int > &solution)

  Mutates solution vector.
- std::pair< std::vector< int >, std::vector< int > > selection (const std::vector< std::vector< int > > &population, const std::vector< Item > &items, double weightLimit)

  Selects two individuals from a population using tournament selection.
- std::pair< std::vector< int >, std::vector< int > > crossover (const std::vector< int > &parent1, const std↩ ::vector< int > &parent2)

  Performs crossover between two parent solutions to produce two offspring.

5.1.1 Detailed Description

File containing necessary functions for program to work.

Definition in file functions.cpp.

5.1.2 Function Documentation

5.1.2.1 calculateFitness()

```
double calculateFitness (
            const std::vector< int > & solution,
            const std::vector< Item > & items,
            double & weightLimit )
```

Calculates the fitness of a solution using a list of items and a weight limit.

Parameters

solution	Vector representing the solution to evaluate.
items	Vector of items.
weightLimit	The maximum weight allowed for the solution.

Returns

The fitness value of the solution.

Definition at line 40 of file functions.cpp.

5.1.2.2 crossover()

```
std::pair< std::vector< int >, std::vector< int > > crossover (
            const std::vector< int > & parent1,
            const std::vector< int > & parent2 )
```

Performs crossover between two parent solutions to produce two offspring.

Parameters

parent1	The first parent solution.
parent2	The second parent solution.

Returns

Pair of offspring solutions produced by performing crossover between the parent solutions.

Definition at line 106 of file functions.cpp.

5.1.2.3 loadItems()

```
std::vector< Item > loadItems (
             const std::string & fileName )
```

Loads a list of items from a file and returns them as a vector.

Parameters

fileName	Input file name as a string

Returns

Vector of items loaded from the file.

Definition at line 20 of file functions.cpp.

5.1.2.4 mutate()

```
void mutate (
             std::vector< int > & solution )
```

Mutates solution vector.

Parameters

solution	Solution vector to mutate.

Definition at line 60 of file functions.cpp.

5.1.2.5 selection()

```
std::pair< std::vector< int >, std::vector< int > > selection (
             const std::vector< std::vector< int > > & population,
             const std::vector< Item > & items,
             double weightLimit )
```

Selects two individuals from a population using tournament selection.

Parameters

population	Vector of vectors representing the population of solutions.
items	Vector of items used to evaluate the solutions.
weightLimit	The maximum weight allowed for the solutions.

Returns

Pair of vectors representing the selected individuals.

Definition at line 79 of file functions.cpp.

5.2 functions.cpp

Go to the documentation of this file.

```
00001
00005 #include <iostream>
00006 #include <fstream>
00007 #include <random>
00008 #include <algorithm>
00009 #include <chrono>
00010 #include <cmath>
00011 #include <vector>
00012 #include <string>
00013 #include "structures.h"
00020 std::vector<Item> loadItems(const std::string& fileName) {
00021     std::vector<Item> items;
00022     std::ifstream file(fileName);
00023     Item item;
00024
00025     while (file » item.name » item.weight » item.value) {
00026         items.push_back(item);
00027     }
00028
00029     file.close();
00030     return items;
00031 }
00040 double calculateFitness(const std::vector<int>& solution, const std::vector<Item>& items, double&
      weightLimit) {
00041     double weight = 0.0;
00042     int value = 0;
00043
00044     for (int i = 0; i < solution.size(); i++) {
00045         if (solution[i] == 1) {
00046             weight += items[i].weight;
00047             value += items[i].value;
00048         }
00049     }
00050     if (weight > weightLimit) {
00051         value = 0;
00052     }
00053     return value;
00054 }
00060 void mutate(std::vector<int>& solution) {
00061     std::random_device rd;
00062     std::mt19937 generator(rd());
00063     std::uniform_real_distribution<double> distribution(0.0, 1.0);
00064
00065     for (int i = 0; i < solution.size(); i++) {
00066         if (distribution(generator) < mutationRate) {
00067             solution[i] = 1 - solution[i];
00068         }
00069     }
00070 }
00079 std::pair<std::vector<int>, std::vector<int» selection(const std::vector<std::vector<int»& population,
00080     const std::vector<Item>& items, double weightLimit) {
00081     std::random_device rd;
00082     std::mt19937 generator(rd());
00083     std::uniform_int_distribution<int> distribution(0, population.size() - 1);
00084
00085     std::vector<int> parent1 = population[distribution(generator)];
00086     std::vector<int> parent2 = population[distribution(generator)];
00087
00088     // Choosing two best solutions
00089     if (calculateFitness(parent1, items, weightLimit) < calculateFitness(parent2, items, weightLimit))
      {
00090         parent1 = population[distribution(generator)];
00091     }
00092     else {
00093         parent2 = population[distribution(generator)];
00094     }
00095
00096     return std::make_pair(parent1, parent2);
00097 }
00106 std::pair<std::vector<int>, std::vector<int» crossover(const std::vector<int>& parent1, const
      std::vector<int>& parent2) {
```

```
    int crossoverPoint = rand() % parent1.size();
    std::vector<int> offspring1(parent1.size());
    std::vector<int> offspring2(parent2.size());
    for (int i = 0; i < crossoverPoint; i++) {
        offspring1[i] = parent1[i];
        offspring2[i] = parent2[i];
    }
    for (int i = crossoverPoint; i < parent1.size(); i++) {
        offspring1[i] = parent2[i];
        offspring2[i] = parent1[i];
    }
    return std::make_pair(offspring1, offspring2);
}
```

5.3 main.cpp File Reference

Main function of the program.

```
#include "structures.h"
#include "functions.cpp"
```

Functions

- int main (int argc, char ∗∗argv)

5.3.1 Detailed Description

Main function of the program.

Author

Kacper Rebosz

Date

2023-02-15

Definition in file main.cpp.

5.3.2 Function Documentation

5.3.2.1 main()

```
int main (
            int argc,
            char ** argv )
```

Definition at line 24 of file main.cpp.

5.4 main.cpp

Go to the documentation of this file.

```
#include "structures.h"
#include "functions.cpp"

int main(int argc, char** argv) {

    if (argc < 2) {
        std::cout « std::endl;
        std::cout « "Switches in program :" « std::endl;
        std::cout « "-i name.txt" « " -- input file with a set of items" « std::endl;
        std::cout « "-o name.txt" « " -- output file with the best solutions in all generations" « std::endl;
        std::cout « "-c value" « " -- knapsack capacity" « std::endl;
        std::cout « "-g value" « " -- number of generations" « std::endl;
        std::cout « "-n value" « " -- number of individuals in a generation" « std::endl;
        return 1;
    }
    bool isEverythingOk = true;
    std::string outputFilename,itemsInput;
    double weightLimit;
    int sizeOfPopulation = 100;
    int numberOfGenerations = 100; // Number of iterations

    for (int i = 1; i < argc; i++) {
        std::string arg = argv[i];
        if (arg == "-i" && i + 1 < argc) {
            itemsInput = argv[i + 1];
        }else if (arg == "-o" && i + 1 < argc) {
            outputFilename = argv[i + 1];
        }
        else if (arg == "-g" && i + 1 < argc) {
            if (std::stoi(argv[i + 1]) < 1) {
                std::cout « "Number of generations provided is invalid. Provide [int] number higher than 0" « std::endl;
                isEverythingOk = false;
            }
            else {
                numberOfGenerations = std::stoi(argv[i + 1]);
            }
        }
        else if (arg == "-c" && i + 1 < argc) {
            if (std::stod(argv[i + 1]) <1) {
                std::cout«"Capacity provided is invalid. Provide number higher than 0"«std::endl;
                isEverythingOk = false;
            }
            else {
                weightLimit = std::stod(argv[i + 1]);
            }
        }
        else if(arg=="-n"&&i+1<argc) {
            if(std::stoi(argv[i+1])<1){
                std::cout « "Number of individuals in a generation provided is invalid. Provide [int] number higher than 0" « std::endl;
                isEverythingOk = false;
            }
            else {
                sizeOfPopulation = std::stoi(argv[i + 1]);
            }
        }
    }
    if (isEverythingOk == false) {
        exit(0);
    }
    // Loading items from file
    std::vector<Item> items = loadItems(itemsInput);

    std::random_device rd;
    std::mt19937 generator(rd());
    std::uniform_int_distribution<int> distribution(0, 1);

    // Initialization of the population
    std::vector<std::vector<int» population(sizeOfPopulation, std::vector<int>(items.size()));
    for (int i = 0; i < sizeOfPopulation; i++) {
        for (int j = 0; j < items.size(); j++) {
            population[i][j] = distribution(generator);
        }
    }

    std::ofstream outputFile(outputFilename);
    // Main function loop
    for (int i = 0; i < numberOfGenerations; i++) {
        // Selection of two solutions by using tournament selection
        std::pair<std::vector<int>, std::vector<int» parents = selection(population, items, weightLimit);

        // Crossover the two solutions to produce a new solution
        std::vector<int> offspring(items.size());
        for (int j = 0; j < items.size(); j++) {
            offspring[j] = (j < items.size() / 2) ? parents.first[j] : parents.second[j];
        }

        // Mutation of new solution
        mutate(offspring);

        // Replacing the worst solution in the population with the new solution
        int worstIndex = 0;
        for (int j = 1; j < sizeOfPopulation; j++) {
            if (calculateFitness(population[j], items, weightLimit) <
                calculateFitness(population[worstIndex], items, weightLimit)) {
                worstIndex = j;
            }
        }
        population[worstIndex] = offspring;

        // Output the best solution for this generation
        double bestFitness = -1.0;
        std::vector<int> bestSolution;
        for (int j = 0; j < sizeOfPopulation; j++) {
            double fitness = calculateFitness(population[j], items, weightLimit);
            if (fitness > bestFitness) {
                bestFitness = fitness;
                bestSolution = population[j];
            }
        }

        outputFile « "Generation " « i+1 « ": ";
        double totalWeight = 0.0;
        double totalValue = 0.0;
        for (int j = 0; j < bestSolution.size(); j++) {
            if (bestSolution[j] == 1) {
                outputFile « items[j].name « " (";
                outputFile « "weight: " « items[j].weight « ", ";
                outputFile « "value: " « items[j].value « ") ";
                totalWeight += items[j].weight;
                totalValue += items[j].value;
            }
        }
        outputFile « "Total weight: " « totalWeight « ", Total value: " « totalValue « std::endl;
    }

    // Finding the best solution in the population
    int bestIndex = 0;
    for (int i = 1; i < sizeOfPopulation; i++) {
        if (calculateFitness(population[i], items, weightLimit) >
            calculateFitness(population[bestIndex], items, weightLimit)) {
            bestIndex = i;
        }
    }
    std::vector<int> bestSolution = population[bestIndex];

    // Printing items in the best solution
    double weight = 0.0;
    int value = 0;
    outputFile « "Best solution: " « std::endl;
    for (int i = 0; i < bestSolution.size(); i++) {
        if (bestSolution[i] == 1) {
            outputFile « "  " « items[i].name « " (weight: " « items[i].weight « ", value: " « items[i].value « ")"
                « std::endl;
            weight += items[i].weight;
            value += items[i].value;
        }
    }
    outputFile « "Total weight: " « weight « std::endl;
    outputFile « "Total value: " « value « std::endl;
    outputFile.close();
}
```

5.5 structures.h File Reference

File that contains the necessary structure definitions for the program to work.

```
#include <vector>
#include <string>
```

Classes

- struct Item

  Struct representing an item in knapsack problem.

Macros

- #define STRUCTURES_H

Variables

- const double mutationRate = 0.05

5.5.1 Detailed Description

File that contains the necessary structure definitions for the program to work.

Definition in file structures.h.

5.5.2 Macro Definition Documentation

5.5.2.1 STRUCTURES_H

```
#define STRUCTURES_H
```

Definition at line 4 of file structures.h.

5.5.3 Variable Documentation

5.5.3.1 mutationRate

```
const double mutationRate = 0.05
```

The mutation rate

Definition at line 25 of file structures.h.

5.6 structures.h

Go to the documentation of this file.

```
#pragma once

#ifndef STRUCTURES_H
#define STRUCTURES_H
#include <vector>
#include <string>

struct Item {
    std::string name;
    double weight;
    int value;
};

const double mutationRate = 0.05;
#endif
```