A Genetic Algorithm for the Number Partitioning Problem

H.S. de Hoop (S5303893) & N. Trigonis (S5991064)
H.S.de.Hoop@student.rug.nl & N.Trigonis@student.rug.nl

November 3, 2024

1 Problem description

Given a multiset S of n integers, divide the values into k subsets so that the union of the subsets equals S. This is the multiway number partitioning problem.

2 Problem analysis

As a well known NP-hard problem, time complexity for exact, deterministic solutions is exponential. Since for large inputs this problem can take incredibly long, approximate solution algorithms have been developed instead. A simple but good one of those is the greedy heuristic algorithm, where set S gets sorted from largest to smallest value and then gets distributed from left to right to the subset with the smallest sum of values. This algorithm is deterministic and solves the problem quite fast, however it is not always accurate. In an attempt to further explore possible approximate solutions, we have decided to take on the probabilistic approach and develop a genetic algorithms. What distinguishes this solution from others is that it finds an approximate solution by stochastic means, which means it could possibly find a perfect solution where a deterministic solution always fails.

Fundamental components for the problem:

- A set S containing n integers.
- An amount k of subsets to distribute the integers.

As we're making a genetic algorithm we also need these components:

- A chromosome, consisting of an array of integers indicating what subset it belongs to.
- Various functions for alterating the chromosome, such as mutation, crossover, etc.
- A way to determine fitness, and a selection mechanism.

Having implemented these components, it is only necessary to continuously select until a terminating condition has been reached. This is either when a solution has been found or when no better solution has been found. Possible terminating conditions:

- When the total difference between all subsets is less than 1, and thus a perfect solution has been found.
- It's been a long time since a solution was found.
- A predetermined time has been reached.

3 Design

Various variables and data structures are declared globally. This is because they are accessed by most functions. Before everything else, the random number generator gets seeded with the current Unix time. This should be adequate for normal use. The input is read; first the number k of subsets and the amount n of integers in the set, into `subsets` and `numBlocks` respectively. Then the value of `chromLength` is set to `numBlocks` and memory allocated to `**generation` and `*blocks`. The actual integers are read right after into `blocks`. `initialize()` is called. This calculates the total sum of the integers and then also introduces a full fresh generation. Now after creating variables `gen`, `drift`, `oldDev`, and `newDev`, the main loop can start.

When reading input, these options are also processed:

- Input can be passed in through standard input or a file by calling it as an argument, or it can be generated randomly with the argument rand [subsets] [numBlocks] [min] [max]
- drift can be set at the beginning of the program with -d x, and popSize with -p x.
- Various print options: -pg; print the final generation, -t; print the elapsed time,
- Option -g, first try the greedy heuristic, if it hasn't found an exact solution, make its result an individual in the generation and then start the genetic loop.

The main loop of the program can be considered the genetic algorithm. First, the generation gets ranked and the lowest average deviation gets stored in `newDev`. If `newDev` is better than the old deviation `oldDev`, `oldDev` gets set to `newDev` and `drift` resets to 0 and if the options are right, the current gen and its associated deviation get printed. Else, `drift` increments. Now we are at the selection process of the algorithm. As our implementation is elitist, the first ranked individual never gets altered. At first, selection is quite strict, but as `drift` increases, mutations are increased. This is to increase diversity as less and less improvements get found over time, as the chromosome might settle into local optima. After the selection process, the loop ends, and if `newDev * numBlocks` has become less than or equal to 1.0, or `drif` has become `maxDrift` it terminates.

Now that the main loop has ended the final phase of the program starts, which is simply printing the final values. First the chromosome gets printed in base 64. Then the full sum of all subsets gets printed in the following format: Set [subset]: [sum of integers]=[result]; dev: [deviation this set has from the theoretical optimum] when this has been done for every subset, the theoretical optimal, average deviation, lowest sum value and highest value get printed on one line. The generation and Iteration get printed, and finally the time gets printed in :minutes:seconds, and then in ticks.

4 Program code

Also can be found on github.com/HSdeH/parprob. Includes associated files.

parprob.c

```
/*
 * by Rik de Hoop & Nikolaos Trigonis
 * A Genetic Algorithm for the Partition Problem
 * AKA: parprob.c
 */

#include "parprob.h"

#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int popSize = 10;
int maxDrift = 10000;
int subsets;
int numBlocks, chromlength;
int totalHeight;
int *blocks;
int **generation;

int main(int argc, char **argv) {
   Options options = {1}; // options, mostly for printing results
   srand(time(NULL));
   readInput(argc, argv, &options);
   initialize();
   if (options.greedy) {
      greedy(generation[0]);
   }
   int *ranks = malloc(popSize * sizeof(int));
   double oldDev = -1, newDev, mutationFactor;
   int gen = 0, drift = 0, winner;
   clock_t start = clock(), end;
   // main loop
   newDev = rankGen(ranks);
   while ((drift < maxDrift) &&
         (newDev * numBlocks > 1.0)) { // continue until a solution has been
                                       // found or maxDrift has been reached
      // rank the individuals and get the lowest deviation
      newDev = rankGen(ranks);
      if (oldDev == newDev) {
         drift++;
      } else {
         if (options.improvement || options.defaultOptions) {
            printf("Gen %-7d: %.3f\n", gen,
                  newDev); // each time an improvement has been found, print
                           // the generation and current improvement
         }
         drift = 0;
         oldDev = newDev;
      }
```

```
      mutationFactor = (double)drift / (double)maxDrift;
      // change the new generation
      for (int i = popSize / 2; i < popSize; i++) {
         winner = (rand() > RAND_MAX / 3)
                    ? 0
                    : 1; // pick either the best or the second best
         recombine(generation[ranks[i]], generation[ranks[winner]]);
      }
      for (int i = 1; i < popSize; i++) {
         mutate(generation[ranks[i]],
                2 * mutationFactor); // this allows the mutation to increase
                                     // to 25% of genes as drift increases
      }
      gen++;
   }
   end = clock();
   // print everything in options
   fullPrint(&options, ranks, newDev, gen, drift, end - start);
   // free all memory
   free(ranks);
   free(blocks);
   for (int i = 0; i < popSize; i++) {
      free(generation[i]);
   }
   free(generation);
   return 0;
}

// reads heights or generates them
void readInput(int argc, char **argv, Options *options) {
   bool input = false;
   int a = 1;
   // argument loop, I don't know of a better way to do this
   while (a != argc) {
      if (!strcmp(argv[a], "-pg")) {
         options->printGen = true;
         options->defaultOptions = false;
      } else if (!strcmp(argv[a], "-t")) {
         options->printTime = true;
         options->defaultOptions = false;
      } else if (!strcmp(argv[a], "-g")) {
         options->greedy = true;
      } else if (!strcmp(argv[a], "-p")) {
         popSize = atoi(argv[a + 1]);
         a++;
      } else if (!strcmp(argv[a], "rand")) {
         if (!input) {
            // instead of giving the input yourself, generate them randomly with
            // as arguments the number of subsets, number of blocks, min block
            // heigh, and max block height, also saves it to a /rand folder, if
            // it doesn't exit, fails
            input = true;
            char path[20];
            snprintf(path, sizeof(path), "rand/%d.in", (int)time(NULL));
            FILE *file = fopen(path, "w");
            if (file == NULL) {
               printf("failure to create %s\n", path);
               exit(EXIT_FAILURE);
```

```
            } else {
               printf("created file %s\n", path);
            }
            subsets = atoi(argv[a + 1]);
            numBlocks = atoi(argv[a + 2]);
            blocks = malloc(sizeof(int) * numBlocks);
            fprintf(file, "%d %d\n", subsets, numBlocks);
            int randMin = atoi(argv[a + 3]), randMax = atoi(argv[a + 4]) + 1;
            for (int i = 0; i < numBlocks; i++) {
               blocks[i] = (rand() % (randMax - randMin)) + randMin;
               fprintf(file, "%d ", blocks[i]);
            }
            fclose(file);
         }
         a = a + 4;
      } else if (!strcmp(argv[a], "-d")) {
         maxDrift = atoi(argv[a + 1]);
         a++;
      } else {
         if (!input) {
            input = true;
            FILE *inputFile = fopen(argv[a], "r");
            if (inputFile == NULL) {
               printf("%s is not a file\n", argv[a]);
               exit(EXIT_FAILURE);
            }
            readFile(inputFile);
            fclose(inputFile);
         }
      }
      a++;
   }
   if (!input) {
      readFile(stdin);
   }
}

// reads file or stdin, to prevent code duplication
void readFile(FILE *input) {
   (void)fscanf(input, "%d ", &subsets);
   (void)fscanf(input, "%d ", &numBlocks);
   blocks = malloc(sizeof(int) * numBlocks);
   for (int i = 0; i < numBlocks; i++) {
      (void)fscanf(input, "%d ", &blocks[i]);
   }
}

// fills all chromosomes of the first generation, and initializes values
void initialize() {
   int total = 0;
   chromlength = numBlocks;
   generation = malloc(sizeof(int *) * popSize);
   for (int i = 0; i < popSize; i++) {
      generation[i] = malloc(sizeof(int) * chromlength);
   }
   for (int i = 0; i < numBlocks; i++) {
      total += blocks[i];
   }
```

```c
    totalHeight = total;
    for (int i = 0; i < popSize; i++) {
        introduce(generation[i]);
    }
}

// return a random tower index, divide RAND_MAX by towers and divide
// rand() by that to get a random number, as this method has a low chance of
// getting towers, in case this happens the tower index decrements
int towerRand() {
    int tower = rand() / (RAND_MAX / subsets);
    return tower == subsets ? tower - 1 : tower;
}

// returns the lowest difference between heights and ranks the individuals
double rankGen(int *ranks) {
    double temp;
    double *rankedDifs = malloc(popSize * sizeof(double));
    for (int i = 0; i < popSize; i++) {
        ranks[i] = i;
    }
    for (int i = 0; i < popSize; i++) {
        rankedDifs[i] = deviation(generation[i]);
    }
    // this probably could have been implemented way better
    for (int i = 0; i < popSize - 1; i++) {
        for (int j = i + 1; j < popSize; j++) {
            if (rankedDifs[i] > rankedDifs[j]) {
                temp = rankedDifs[i];
                rankedDifs[i] = rankedDifs[j];
                rankedDifs[j] = temp;

                temp = ranks[i];
                ranks[i] = ranks[j];
                ranks[j] = temp;
            }
        }
    }
    temp = rankedDifs[0];
    free(rankedDifs);
    return temp;
}

// returns the average deviation of a chromosome,
// the lower the deviation the higher the fitness
double deviation(int *chromosome) {
    int *towers = calloc(sizeof(int), subsets);
    double optimum = (double)totalHeight / (double)subsets;
    double deviation = 0.0;
    // build every tower
    for (int g = 0; g < chromlength; g++) {
        towers[chromosome[g]] += blocks[g];
    }
    // then add up the deviation from the optimum for every tower
    for (int t = 0; t < subsets; t++) {
        deviation += fabs((double)towers[t] - optimum);
    }
    free(towers);
```

```
    return deviation / subsets;
}

// a uniform crossOver, might be a bit slower, forces half of origin into target
void recombine(int *target, int *origin) {
    for (int g = 0; g < chromlength; g++) {
        if (rand() > RAND_MAX / 2) {
            target[g] = origin[g];
        }
    }
}

// after a random index, swap the values of both chromosomes, replaced by
// recombine, but left in to see how it was constructed
void crossOver(int *chrom1, int *chrom2) {
    bool temp;
    int start = (rand() % (chromlength - 1)) + 1;
    for (int i = start; i < chromlength; i++) {
        temp = chrom1[i];
        chrom1[i] = chrom2[i];
        chrom2[i] = temp;
    }
}

// mutates 5% of genes by default, changed yes by factor
void mutate(int *chromosome, double factor) {
    int idx ,rndm, muts;
    // mutate 5% of the genes, multiplied by factor, and at least 1
    int mutations = 1 + factor;
    for (int i = 0; i < mutations; i++) {
        idx = rand() % chromlength;
        do {
            rndm = towerRand();
        } while (rndm == chromosome[idx]);
        chromosome[idx] = rndm;
    }
}

// fill a chromosome with random values
void introduce(int *chromosome) {
    for (int g = 0; g < chromlength; g++) {
        chromosome[g] = towerRand();
    }
}

// replace target chromosome with origin
void replace(int *target, int *origin) {
    for (int g = 0; g < chromlength; g++) {
        target[g] = origin[g];
    }
}

// print final results, depending on options
void fullPrint(Options *options, int *ranks, double deviation, int gen, int cnt,
               int time) {
    if (options->printChrom || options->defaultOptions) {
        printChrom(ranks);
    }
```

```
    if (options->printFullSum || options->defaultOptions) {
        printFullSum(ranks, deviation);
    }
    if (options->printGen) {
        printGen();
    }
    if (options->printGenIts || options->defaultOptions) {
        printf("Generation: %d; Iteration: %d\n", gen - cnt, gen);
    }
    if (options->printTime || options->defaultOptions) {
        printTime(time);
    }
}

// print the sum and information about the towers
void printFullSum(int *ranks, double deviation) {
    int first, sum;
    int max = totalHeight / subsets, min = max;
    for (int s = 0; s < subsets; s++) {
        sum = 0;
        first = 1;
        printf("Set %c:\t", toBase64(s));
        for (int b = 0; b < chromlength; b++) {
            if (generation[ranks[0]][b] == s) {
                if (first) {
                    first = 0;
                    printf("%d", blocks[b]);
                } else {
                    printf("+%d", blocks[b]);
                }
                sum += blocks[b];
            }
        }
        if (sum < min) {
            min = sum;
        }
        if (sum > max) {
            max = sum;
        }
        printf("=%d; dev: %.3f\n", sum,
               (double)sum - ((double)totalHeight / (double)subsets));
    }
    printf("Optimal: %.3f; Average deviation: %.3f; Min: %d; Max: %d\n",
           (double)totalHeight / (double)subsets, deviation, min, max);
}

// print the elapsed time in :mm:ss and ticks
void printTime(clock_t time) {
    int s = ((double)(time) / CLOCKS_PER_SEC);
    printf(":%02d:%02d %dt\n", s / 60, s % 60, (int)time);
}

// print the all the individual chromosomes of the current generation
void printGen() {
    for (int i = 0; i < popSize; i++) {
        for (int j = 0; j < chromlength; j++) {
            printf("%c", toBase64(generation[i][j]));
        }
```

```
      printf("\n");
   }
}
// print a chromosome
void printChrom(int *ranks) {
   printf("Chromosome: ");
   for (int i = 0; i < chromlength; i++) {
      printf("%c", toBase64(generation[ranks[0]][i]));
   }
   printf("\n");
}

// implemented to make the printed chromosome better capable of handling lots of
// towers
char toBase64(int n) {
   char base64[] = {
       "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ+/"};
   return base64[n % 64];
}

/*the greedy algorithm and associated compare function*/
int compare(const void *a, const void *b) { return (*(int *)b - *(int *)a); }
void greedy(int *chrom) {
   int *sums = calloc(sizeof(int), subsets);
   qsort(blocks, numBlocks, sizeof(int), compare);

   for (int g = 0; g < chromlength; g++) {
      chrom[g] = 0;
      for (int j = 1; j < subsets; j++) {
         if (sums[chrom[g]] > sums[j]) {
            chrom[g] = j;
         }
      }
      sums[chrom[g]] += blocks[g];
   }
   free(sums);
}
```

5 Test results

The program was tested on a variety of inputs. These were both created and randomly generated. As a probabilistic program output is not always the same.

Given a simple input of file input/87654.in

```
$ ./parprob input/87654.in
```

The default output is generated. For the first few lines, the program will print the number of the generation an improvement was found and print the associated average deviation. After it will print the chromosome, the sum of every set and its individual deviation, and further information about the tower. Finally it will print the latest generation that was better than the previous and the generation the program quit at. At last it will print the time it took, which in this case was 60 ticks.

```
Gen 0     : 2.000
Gen 4     : 0.000
Chromosome: 11000
Set 0: 6+5+4=15; dev: 0.000
Set 1: 8+7=15; dev: 0.000
Optimal: 15.000; Average deviation: 0.000; Min: 15; Max: 15
Generation: 5; Iteration: 5
:00:00    60t
```

Sometimes, an input might not be able to be perfectly distributed.

```
$ ./parprob input/19x1plus20.in
```

When this happens, the program might print this result:

```
Gen 0     : 7.500
... [other generation]
Gen 11    : 0.500
Chromosome: 11111111111111111110
Set 0: 20=20; dev: 0.500
Set 1: 1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1=19; dev: -0.500
Optimal: 19.500; Average deviation: 0.500; Min: 19; Max: 20
Generation: 12; Iteration: 10012
:00:00    51270t
```

`maxDrift` and `popSize` can be set with `-d [x]` and `-p [x]`. These influence how long the program will try to look for improvements and how much diversity can be saved (to a limited extent) respectively.

```
$ ./parprob input/87654.in -p 1000 -d 0
```

This program always immediately terminates, but since the amount of initial guesses is so high, for a simple problem like this it will (almost) always get a correct answer anyway.

The greedy algorithm can be emulated by using the greedy switch `-g` and setting `popSize` to 1 and `maxDrift` to 0. This might be useful for testing.

```
$ ./parprob input/2x1000x87654.in -p 1 -d 0 -g
```

Given a more dificult problem that in theory could be evenly distributed, but didn't because the algorithm does not converge:

```
$ ./parprob input/50x87654.in -d 100000
```

Notice the base-64 integer representation.

```
...
Chromosome: cb969b5fajd4111e2i6i2gh884gajafc86057330d0ji37e9hh
Set 0: 4+4+7=15; dev: 0.000
Set 1: 6+5+4=15; dev: 0.000
...[set 2 to 17 (h)]
Set i: 6+4+5=15; dev: 0.000
Set j: 4+5+6=15; dev: 0.000
Optimal: 15.000; Average deviation: 0.200; Min: 14; Max: 16
Generation: 88744; Iteration: 188744
:00:01    1632584t
```

Usage of rand. Generates 100 integers with values between 1 and 100000 and divides them between 11 subsets.

```
$ ./parprob rand 11 100 1 100000 -d 1000000
```

The obviously randomly generated set.

```
created file rand/1730585199.in
Gen 0     : 86431.603
...
Gen 1267692: 126.992
Chromosome: 4351803240832a6412a6278777264343491162256197aa7545987165087623a69a5561869a059
7080983a119761153960418
Set 0: 69405+66601+30323+83231+37862+94602+58471=440495; dev: -119.636
...
Set a: 83644+48003+475+35913+39829+85128+96556+51158=440706; dev: 91.364
Optimal: 440614.636; Average deviation: 126.992; Min: 440391; Max: 441107
Generation: 1267693; Iteration: 2267693
:00:24    24072644t
```

Typically, the more integers are generated, the better the initial guesses.

6 Evaluation

Some obvious and less obvious flaws still remain in the program. The most blatant one being that diversity is not preserved well enough, resulting in the program getting easily stuck in local optima. If the program had to be reworker, it could be possible to ensure both a more complex selection algorithm, and also introduce a mechanic saving different possible solutions from destroying eachother. Such an improvement could be the introduction of "islands", allowing each island to find its own solution before comparing it to the other islands. Although, this could also be recreated by running the program multiple times.

After testing multiple options, we chose a relatively simple selection algorithm. Rather than replacing individuals, the one of the top 2 individuals recombines itself with the bottom half of individuals. Diversity is preserved slightly better this way. Also, instead of using the suggested crossover with the head and tail being swapped our crossover `recombine()` instead just takes randomly half of one chromosome genes and places those in the other chromosome. This was done mostly to make it function better with the greedy heuristic, which sort the full set of integers.

As one of the goals in this project was to test how the algorithm would work for certain inputs and to compare it to the greedy algorithm, it would have been better to save the results to another file. This way, graphs could have been made to more easily compare results. As for how the genetic algorithm compares to the greedy one, it for sure is always slower, but in certain cases where the greedy algorithm can't find a perfect solution, the genetic one might find a better result. Supplementing the greedy algorithm with the genetic algorithm doesn't seem to work much better than just calling the genetic algorithm by itself multiple times.

Our program does what it was supposed to do. It has its flaws, and for relatively complex inputs it's guaranteed to never come to an optimal solution. Despite this, it provides a somewhat quick way to find solutions that can be better than other approximate algorithms.