A Genetic Algorithm for the Number Partitioning Problem

H.S. de Hoop & N. Trigonis
H.S.de.Hoop@student.rug.nl & N.Trigonis@student.rug.nl

October 26, 2024

1 Problem description

Given a multiset S of n integers, divide the values into k subsets as evenly as possible. 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.

Components needed for the genetic algorithm:

- 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.

Possible options for fitness:

- The average deviation of all towers from the optimally distributed height.
-

Terminating conditions:

- When the difference between all subsets is none, and thus a perfect solution has been found.

3 Design

Before everything else, seed the random number generator with the current Unix time. Read the input; first the number k of subsets and the amount n of integers in the set, into `subsets` and `numBlocks` respectively. Then set the value of `chromLength` to `numBlocks` and allocate memory to `**generation` and `*blocks`. Call the initialize function. 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. 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, selection gets more and more permissive, increasing mutation, and individuals even get replaced by new randomly generated ones. 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` has become 0, or `drif` has become `maxDrif` it terminates.

Being a probabilistic algorithm, multiple auxiliary functions were written to analyze the results.

First the points A, B, P and Q are read from the standard input and stored (in the variables `xA`, `yA`, `xB`, `yB`, `xP`, `yP`, `xQ`, and `yQ`). Next, the parameters of the line through A and B are computed (`deltax`, `deltay`, and `c`). Using these, we determine whether P and Q are split up by the line trough A and B (stored in `PQdivided`).

After this computation, we perform the same for the other line segment through P and Q. Note that the variables `deltax`, `deltay` and `c` were used to compute `PQdivided`, and can now be reused in the computation of `ABdivided`, which denotes whether the points A and B are split up by the line trough P and Q.

Finally the answer is computed from the variables `PQdivided` and `ABdivided`. It is presented to the user by printing it on the standard output.

4 Program code

parprob.c

```
#include <stdio.h>
#include <stdlib.h>

/* (C) Arnold Meijster & Doina Bucur, september 2015:
*/

int main(int argc, char *argv[]) {
    int PQdivided; /* Are P and Q divided through AB? */
```

```c
    int ABdivided; /* Are A and B divided through PQ? */
    int xA, yA; /* coordinates of point A */
    int xB, yB; /* coordinates of point B */
    int xP, yP; /* coordinates of point P */
    int xQ, yQ; /* coordinates of point Q */
    int deltax, deltay, c; /* parameters of the equation */

    /* input coordinates */
    printf("Please enter the x- and y-coordinates, separated by a space.\n");
    printf("Point A: ");
    scanf("%d %d", &xA, &yA);

    printf("Point B: ");
    scanf("%d %d", &xB, &yB);

    printf("Point P: ");
    scanf("%d %d", &xP, &yP);

    printf("Point Q: ");
    scanf("%d %d", &xQ, &yQ);

    /* determine whether P and Q are separated by the line trough A and B */
    deltax = xB - xA;
    deltay = yB - yA;
    c = yA*deltax - xA*deltay;
    PQdivided = ((yP*deltax - xP*deltay - c)*(yQ*deltax - xQ*deltay - c) < 0 );
    /* determine whether A and B are sperated by the line trough P and Q */
    deltax = xQ - xP;
    deltay = yQ - yP;
    c = yP*deltax - xP*deltay;
    ABdivided = ((yA*deltax - xA*deltay - c)*(yB*deltax - xB*deltay - c) < 0 ) +
                ((yA*deltax - xA*deltay - c)*(yB*deltax - xB*deltay - c) < 0 );
    /* print result */
    if (PQdivided && ABdivided) {
       printf("The line segments intersect.\n");
    } else {
       printf("The line segments do not intersect.\n");
    }
    return 0;
}
```

5 Test results

- Input: (normal case with one common point)

  ```
  0 0
  9 9
  4 9
  8 1
  ```

  Output:

  ```
  The lines intersects each other.
  ```

- Input: (normal case without common point)

```
0 0
4 8
9 1
5 8
```

Output:

```
The lines do not intersects each other.
```

- Input: (P on line of AB)

```
0 0
9 9
5 5
9 1
```

Output:

```
The lines do not intersects each other.
```

- Input: (all points on one line)

```
0 0
7 7
5 5
9 9
```

Output:

```
The lines do not intersects each other.
```

- Input (lines with equal start-point and end-point)

```
5 5
5 5
8 2
1 9
```

Output:

```
The lines do not intersects each other.
```

6 Evaluation

Due to our relative simple definition of line segment intersection, the program was rather easy to design and write. For our definition of intersection (i.e. only one common point), the program produces correct results. If you would change the definition to at least one common point, further analysis would be necessary. Then the three different results of the product: positive, negative and zero have to be investigated. Which is a bit harder if both expressions return zero as a result. But luckily our definition is sufficient for this assignment.

We have chosen a geometric approach. A mathematical approach would have been possible as well. Then we would have computed the intersection point given the line equations. In this case you have to take care of multiple intersection points. Moreover, you also need to test whether the calculated intersection point is on the line segments. We think that the geometric approach is a lot easier.