Genetic Partition Problem

Channa Dias Perera, S4400739 (c.dias.perera@student.rug.nl
Ola Dybvadskog, S4530012 (o.dybvadskog@student.rug.nl)

November 4, 2020

Contents

1 Introduction

1.1 Preliminary definitions

Let us first define some auxiliary functions and sets.

We define a set N, which is a set containing all finite sequences of natural numbers.

We define a function $\text{sum} : N \to \mathbb{N}$ such that

$$\text{sum}(s) = \sum_{i=0}^{n} s_i$$

Where s_i is the i^{th} member of s and n is the length of the sequence s.

The initialism a.d.s stands for "*absolute difference of sums*".

1.2 The partition problem

The **partition problem** is defined as such: given a finite sequence s of positive integers, we must determine two sequences d_1 and d_2 such that:

1. The sequences d_1 and d_2 only contain members from the sequence s.
2. Every member in s must also be present in either d_1 or d_2.
3. The sequences d_1 and d_2 do not share any members.
4. Let x_1 and x_2 be finite integer sequences that would satisfy constraints 1, 2 and 3. Then d_1 and d_2 are the two sequences x_1 and x_2, respectively, such that $|\text{sum}(x_1) - \text{sum}(x_2)|$ is minimized.

2 Problem analysis

While the problem is easy to understand, the only algorithm that ensures that a solution is found is to check every possible pair of sequence that satisfies constraints 1, 2 and 3, of which there are 2^n possible pairs to check. This is because of the n members in the sequence, each number has 2 possible sets to be placed into.

Of course, such an algorithm is in $\mathcal{O}(2^n)$, which is intractable for all but the smallest values of n. While there are some optimizations that could be made, due to symmetry, this approach is still $\mathcal{O}(2^n)$.

Instead, we shall approach the problem by using a **genetic algorithm**. Such an algorithm obtains the solution by a simulation of *evolution*. Broadly, this means that a *generation* of *chromosomes* are generated, of which the *stronger* chromosomes propagate (and potentially spread) through to the next generation, and the *weaker* chromosomes *die*, due to *selection pressure*,

In addition, the next generation of chromosomes has:

- A number of *mutations* applied to it.
- A number of recombinations of two chromosomes, called a *cross-over*.

3 Design

3.1 The application of a genetic algorithm to the partition problem

We shall now discuss the specific details of how we applied a genetic algorithm to the partition problem.

We will use m and k to refer to the lengths of sequences d_1 and d_2 respectively. In addition, we will denote $d_{a,i}$ to refer to the i^{th} member of sequence d_a.

Below is a diagram that illustrates the important steps in the algorithm. We will discuss the decisions and "*Any improvements?*" and "*No improvement for a while?*" in section **3.3**.

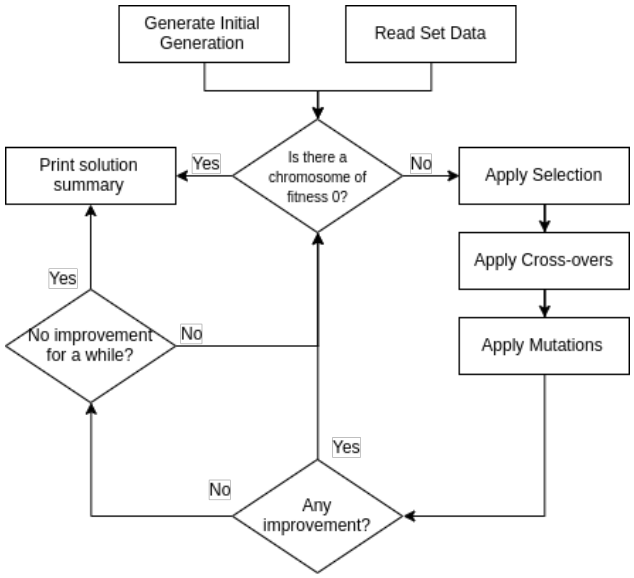

Figure 1: The steps involved in simulating evolution.

3.1.1 Representing a solution in a chromosome

A **chromosome** will represent a particular "solution" to a given input sequence, however the chromosome may not necessarily represent a correct solution.

As there are only two disjoint sequences, and as any given element of the original set may be present in only one sequence, we can represent which sequence a member belongs to with a binary number. Let us represent a chromosome as an n-length sequence c, where n is the length of the original sequence. Any individual element $c_i \in c$ is called a **gene**. Then for any element c_i:

- $c_i = 1$ if the element belongs to d_1
- $c_i = 0$ if the element belongs to d_2

When writing "*a.d.s of the chromosome*", we mean "*absolute difference of sums of the two sequences encoded by the chromosome*".

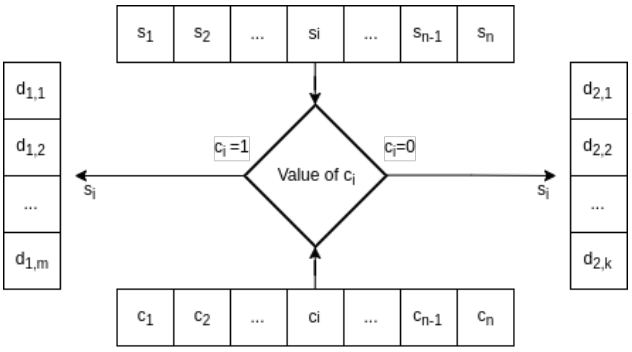

Figure 2: Solution representation as a chromosome.

3.1.2 Selection pressure

A chromosome is stronger if it is **fitter** than another chromosome. A fitter chromosome is one that is closer to the solution. That is, when the a.d.s of the two sequences is as small as possible.

Usually, one would determine a function for fitness such that stronger chromosomes are mapped to larger values of fitness. In our case, there would be an inverse relationship between the a.d.s of a chromosome and the fitness value. However, this would result in greater implementation difficulties as:

- We need to represent a chromosome with an a.d.s of 0 in a special manner, as we cannot divide by 0.
- We have to contend with division, which can be expensive computationally but more importantly, it can lead to the imprecise storage of values.

Instead, we will write our program simply storing the a.d.s as our fitness, keeping in mind that a "lower" fitness yields a stronger chromosome.

To simulate selection pressure, we replace the $0 < r < \frac{n}{2}$ weakest chromosomes with the r fittest chromosomes. Thus, the weaker chromosomes die and the stronger chromosomes propagate.

3.1.3 Mutation and Cross-overs

To **mutate** a chromosome, we shall simply flip one of the bits in the chromosome.

To **cross-over** two chromosomes, we shall determine a cross-over point d, where $0 < d \leq n$. All genes in a position smaller than d will be called the **head** of the chromosome. All genes in an index greater than or equal to d will be called the **tail** of the chromosome. Then, to cross-over two chromosomes, we attach the heads of chromosome 1 and 2 to the tails of chromosome 2 and 1, respectively.

There is the question of when we should preform the mutations and cross-overs. Preliminary testing suggests that the program is more optimal when such changes take place after selection, applied on the whole population, instead of applying them during the selection stage. We suspect that is because a greater variety of changes take place.

To avoid "forgetting" the best solution so far, we will avoid mutating the best chromosome, or performing a cross-over with it.

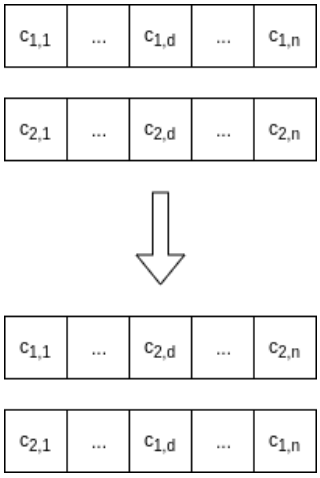

Figure 3: Performing a cross-over. $c_{a,i}$ is the i^{th} member of the chromosome c_a.

3.2 Simulation Configuration

Of course, it's important to consider:

- How many mutations occur when generating a new generation.
- How many cross-overs occur when generating a new generation.
- How many of the weak chromosomes are replaced by the strongest chromosomes.
- The population size of each generation.

If we are too aggressive with the first two parameters, we make it difficult to converge to a solution, as the better solutions are quickly "forgotten". If we are too conservative however, this slows down our algorithm.

With the third parameter, if we are too aggressive, we will reduce the diversity of our chromosome pool and thus lead into a local maximum that might be hard to leave. If we are too conservative, weaker solutions might persist unnecessarily and we won't see changes for a while.

Finally, with the last parameter, we can note that increasing the population would result in fewer iterations being required. This is because the larger the amount of chromosomes present, the greater the chance of a solution chromosome being present (or a chromosome close to it). However, a greater population would result in our algorithm requiring more atomic steps in the first place, and thus, naively increasing it would be foolish.

The determination of the optimal population size is out of the scope of this project and report, since it would greatly complicate the analysis. Thus, we shall use a population size of 10, which we found to be adequate in terms of speed, whilst still providing a correct solution (given the probabilistic nature of the algorithm).

Due to time constraints, we were unable to find a truly optimal configuration. Instead, we tried various configurations till we found one that was efficient enough.

3.3 Implementation Considerations

There are a few details we must consider when implementing the genetic algorithm.

- The non-termination of our program due to randomness
- The non-termination of our program due to the input sequence.

3.3.1 Non-termination due to randomness

There is no guarantee that the algorithm will come to a solution. As all mutations are random, there is no assurance that a mutation will make the chromosome come closer to a solution.

To avoid this, we will have a terminating condition such that after t_1 iterations, we will stop the simulation and report that the best solution so far. It is exceedingly unlikely that this happens however.

3.3.2 Non-termination due to the input sequence

Also, we must avoid naively terminating the algorithm only when the a.d.s of our best chromosome is zero as this can potentially result in a non-terminating program. This is because some sequences cannot be partitioned in such that the a.d.s any two valid finite sequences is zero.

Take for example, a sequence of length n where $n-1$ values are 1 and the other value is n itself. Then, collection including n will have a sum greater or equal to n, but then the other collection will have a value only equal to the number of elements in that collection, which must necessarily be lesser than n!

To avoid this, we will have a terminating condition such that after t_2 iterations, we will stop the simulation and report the best solution found, so far.

Choosing t_2 has a significant impact on our algorithm. Let us say we have an input sequence with a solution that has a non-zero a.d.s. Then, we must necessarily "waste" t_2 iterations after we find our solution (which of course, we cannot know is a solution in advance).

Thus, we are incentivized to minimize t_2. However, we must not be too aggressive with this optimization. If we make t_2 too small, we risk not actually finding our solution, since our solution may not have popped up since our previous best solution. We decided to empirically choose t_2, based on observations of how many iterations it takes to find a solution.

Before we describe how we chose t_2, it's worth making an observation. As $t_2 \to \infty$, we will get better results, and more importantly, we will get more *consistent* results. That is, for a given input, we will arrive at the same minimized a.d.s (though the solution chromosome may be different), as we are more likely to find better solutions, and solutions can only improve up to some level of fitness. Thus, we want to use a t_2 that leads to extremely consistent results.

To do so, we shall use the following methodology:

1. We set $t_2 = 1$
2. We run 100 simulations and keep track of the results
3. If the smallest a.d.s of the 100 simulations does not account for a large percentage, say 90% , of the results, we increase t_2 by a factor of 10 and start again from step 2.

   We want to check the consistency of the *best* solution (as bad solutions may happen to exist in greater number.
4. Let us say our last t_2 is $t_{2,m}$. We repeat step 2 for $t_2 = t_{2,m}/10$ to $t_{2,m}$ in increments of $t_{2,m}/10$.
5. We choose the smallest t_2 such that the smallest a.d.s in the 100 simulations, accounts for 90% of the results.

Thus, we are comfortable that our solution would reach the result 90% of the time.

3.4 Input Validation

We made sure that none of the inputs were below 0.

In addition, we made sure that the total sum of the input sequence was less than `INT_MAX + 1`, so that a potential chromosome that puts all the values in a single set does not cause an integer overflow.

This could potentially be optimized, as such a chromosome always fails, and we could modify our program to avoid such a chromosome. Then, we could increase our input sequence such that we require only the sum of the 19 smallest elements to be less then `INT_MAX + 1`. However, this would require a relatively large refactor of the code (with potentially a performance impact as well), and thus, we have avoided such an optimization.

4 Results

4.1 Determining an upper bound for t_2

Following our methodology as described in section 3.3.2, we found that the best t_2 was 500000, with a percentage similarity of 75%. Whilst a higher percentage similarity would have been desirable, we had to consider the runtime of the algorithm.

Figure 4 is a graph of how the program performs as t_2 increases, on a logarithmic scale scale, showing the performance for smaller values of t_2. We can see we make no gains till around $t_2 = 10000$, at which point we start to see linear gains, till this performance speed up slows down near $t_2 = 500000$, at which point logarithmic behavior is observed, as in Figure 5. This is also partially why we choose a t_2 of 500000.

Please note that `Percentage Similarity` refers to what percentages of the simulation had the smallest a.d.s for the solution.

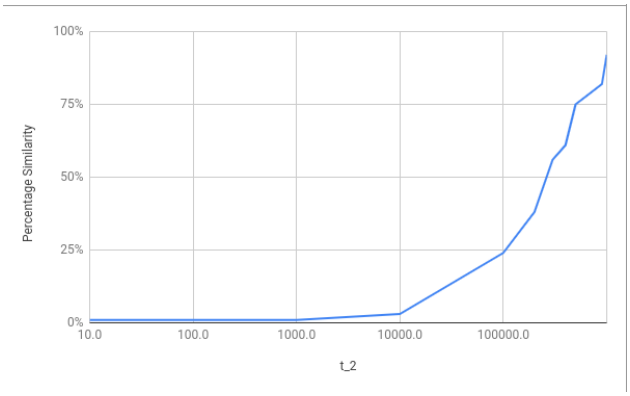

Figure 4: The consistency of solutions, as t_2 increases, on a logarithmic scale

4.2 Summary of the parameters

In Table 4.2 is a summary of all the parameters relevant to our program.

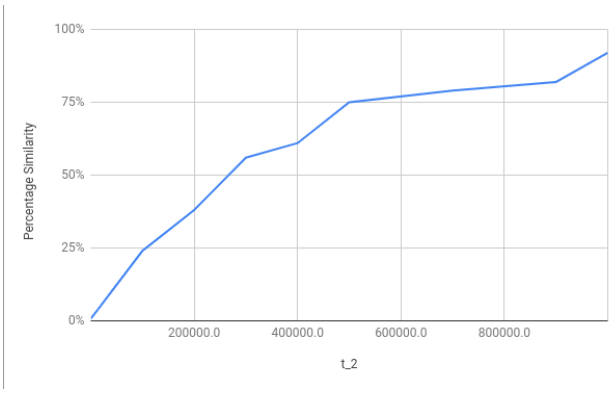

Figure 5: The consistency of solutions, as t_2 increases, on a linear scale

Simulation parameter	Chosen value
Number of mutations	20
Number of cross-overs	4
Number of chromosomes replaced due to selection pressure	2
t_1	5000000
t_2	500000

Table 1: Optimal simulation values

It's worth mentioning that 20 mutations will only mutate up to $\frac{20}{20 \cdot 10} \cdot 100 = 20$ percent of the gene pool. So while this number of mutations may seem to be a lot, it is not that much, in practice.

4.3 Inputs of interest

In this section, we will present some specific input sequences we have passed to the program, in addition to a solution provided by the program as well as the number of iterations it takes for the program to find a solution.

- The first test case we shall consider is **when the a.d.s of the solution is** 0.

  Input:

```
1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

  Output:

```
ZERO'D difference of sums!
CHROMOSOME: 0 1 0 0 1 1 1 1 1 0 0 1 0 1 0 0 0 1 1 0
================================================================================
COLLECTION 1: {2, 5, 1, 2, 3, 4, 2, 4, 3, 4}
sum = 30
COLLECTION 2: {1, 3, 4, 5, 1, 3, 5, 1, 2, 5}
sum = 30
================================================================================
Number of Iterations: 11
```

As expected, we arrived at a solution quickly.

- The second test case we shall consider is **when the a.d.s of the solution is not** 0.

  Input:

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 20
```

  Output:

```
MINIMIZED absolute difference of sums: 2147483609!
CHROMOSOME: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1
================================================================================
COLLECTION 1: {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
sum = 19
COLLECTION 2: {2147483628}
sum = 2147483628
================================================================================
Number of Iterations: 500079
```

  Interestingly, we found the solution after 79 iterations, at which point we had to waste 500000 iterations to make sure it was the best it could find.

- The third test case we shall consider is **when the a.d.s of the solution is large**.

  Input:

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2147483628
```

  Output:

```
MINIMIZED absolute difference of sums: 2147483609!
CHROMOSOME: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1
================================================================================
COLLECTION 1: {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
sum = 19
COLLECTION 2: {2147483628}
sum = 2147483628
================================================================================
Number of Iterations: 500112
```

- The fourth test case we shall consider is **when the all the input numbers are the same**.

  Input:

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

  Output:

```
ZERO'D difference of sums!
CHROMOSOME: 1 0 0 0 0 1 1 1 0 1 0 1 1 0 0 1 0 1 1 0
================================================================================
COLLECTION 1: {1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
sum = 10
COLLECTION 2: {1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
```

```
sum = 10
================================================================================
Number of Iterations: 4
```

A solution should be present when each sequence has an equal number of members. This should arise fairly quickly. However, there's no guarantee the chromosome would have such a solution in its initial generation. That said it should arise fairly quickly.

- The last test case we shall consider is **a random input sequence**.

  We have limited the input sequence such that it is less than the `INT_MAX` (as this would cause our program to report an error).

  Input:

```
86302471 95311612 71080047 104024185 25012517 102115789 75640294 39147762 59645739
     8525419 58834724 61999843 31749416 28778239 4788305 34778650 76690356
    37144878 89340808 14965552
```

  Output:

```
MINIMIZED absolute difference of sums: 262!
CHROMOSOME: 1 0 0 1 1 0 1 0 1 0 1 1 0 0 1 0 1 0 0 0
================================================================================
COLLECTION 1: {86302471, 104024185, 25012517, 75640294, 59645739, 58834724,
    61999843, 4788305, 76690356}
sum = 552938434
COLLECTION 2: {95311612, 71080047, 102115789, 39147762, 8525419, 31749416,
    28778239, 34778650, 37144878, 89340808, 14965552}
sum = 552938172
================================================================================
Number of Iterations: 816938
```

5 Evaluation

We are generally happy with the algorithm, though it is not as optimized as it could be. Running our program for a single simulation could take between 5 to 10 seconds per simulation. Thus to test enough configurations, to find an ideal one, it would take days, at the minimum.

It also is quite frustrating to have to waste so many iterations when finding an "easy" solution, as in test case 2. Potentially, one could scale t_2 by how "difficult" the problem is. We suspect that the difficulty lies in the number of repeated numbers in a sequence.

We are also happy that we know how likely we are to find a solution, which allows the program to effectively be used as a tool. If one is happy to have the correct answer 75% of the time, they can choose to use this program, otherwise, they may choose another program.

6 Future Work

Whilst we concluded that the best place to change the population (via mutations and cross-overs) was after the selection stage, there may perhaps be a better stage to apply such changes. It would be worth exploring such an idea.

In addition, it would be interesting to determine the optimal configuration for an input sequence of arbitrary length. Different sequence lengths might have different optimal configurations.

Lastly, it would be valuable to mathematically derive an optimum set of parameters for the genetic algorithm (as presented in this paper).

A Code

Slight modifications have been made to the program, after data was gathered. These modifications have been cosmetic, where the output was slightly formatted, and thus the data collected is still valid.

List of Files

A.1 Files

```c
/* file     : main.c
 * author   : Channa Dias Perera (c.dias.perera@student.rug.nl)
 *          : Ola Dybvadskog     (o.dybvadskog@student.rug.nl)
 * date     : October 2020
 * version  : 1.0
 */

/* Description:
 * Given two sets of positive integers, we create a pair of disjoint sets.
 * The union of these two sets must be the original set, and the sum of the
 * integers in each set must equal each other.
 * We shall use a genetic algorithm to determine these two sets.
 *
 * Input: _Precisely_ SIZE_ORIGINAL_SET (20) integers, separated by whitespace
 *
 * Output: The solution chromosome, and how the chromosome leads to the best
 * solution it comes to.
 *
 * Note: This program was written as part of the assessment for the course
 * "Introduction to Computer Science".
 */

#include "partitionProblem.h"
#include "simconfig.h"
#include "testing.h"
#include "output.h"

int main(int argc, char *argv[]) {
  time_t t;
  // Set seed for random generator
```

```
  srand((unsigned int) time(&t));

  set_t set;
  chromo_t generation[POP_SIZE];

#if TESTING
  findT2();
#endif

#if !TESTING
  // Grab predefined simulation configuration
  simConfig.numCrossOvers = NUM_CROSS_OVERS;
  simConfig.numMutations = NUM_MUTATIONS;
  simConfig.numChromosReplaced = NUM_CHROMOS_REPLACED;
  simConfig.t2 = MAX_T2;

  // Don't create set programatically, read from input
  getInitialSet(set, false);
  int solDifference;
  int numIterations = simulateEvolution(set, &solDifference);

  printf("Number of Iterations: %d\n", numIterations);
#endif

  return 0;
}
```

File 1: Code to begin the program

```
/* file     : partitionProblem.h
 * author   : Channa Dias Perera (c.dias.perera@student.rug.nl)
 *          : Ola Dybvadskog     (o.dybvadskog@student.rug.nl)
 * date     : October 2020
 * version  : 1.0
 */

/* Description:
 * Headers and constants for the partition problem.
 */
#ifndef PARTITIONPROBLEM_H
#define PARTITIONPROBLEM_H

// LIBRARY IMPORTS
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <limits.h>

#include "simconfig.h"

// Maximum value for input sequence when auto-generating
#define AUTO_GEN_SEQ_MEMBER_MAX (INT_MAX / 20)

// Converge statuses
#define CONVERGING 0
#define SOLUTION_FOUND 1
#define NO_IMPROVEMENT -1
#define PAST_MAX_ITER -2

// Indexes of the best/worst chromosome
#define BEST_CHROMO 0
```

```c
#define WORST_CHROMO (POP_SIZE - 1)

// The worst possible fitness
#define WORST_FITNESS INT_MAX

// Output constants
#define DIVIDER_CHAR '='
#define DIVIDER_LEN 80

// TYPES
typedef bool gene_t;

typedef struct {
  gene_t genes[CHROMO_LENGTH];
  int fitness;
} chromo_t;

typedef int set_t[SIZE_ORIGINAL_SET];
// FUNCTIONS

// Functions to initialize program
void  getInitialSet(set_t set, bool autoCreateSet);

// Main functions for Genetic Algorithm
int   simulateEvolution(set_t set, int *solDifference);
void  makeInitialGenration(set_t set, chromo_t *generation);
void  performSelection(set_t set, chromo_t *generation);
int   converges(
  set_t set,
  chromo_t *generation,
  chromo_t *solChromo,
  int *numIterNoImprov,
  int prevBestFitness
);
void  generateNewGeneration(set_t set, chromo_t *generation);

// Functions to manipulate choromosomes
void  sortChromos(chromo_t *generation);
void  swap(int i, int j, chromo_t *generation);
void  copyChromo(chromo_t *dst, chromo_t src);
void  replaceChromos(
  int *strongChromos,
  int *weakChromos,
  chromo_t *generation,
  int numReplaced
);

// Functions to modify genes
void  generateRandomChromo(set_t set, chromo_t *chromo);
void  mutateSingleGene(set_t set, chromo_t *chromo);
void  chromoCrossOver(set_t set, chromo_t *chromo1, chromo_t *chromo2);

// Functions related to fitness
int   heightOfSet(set_t set, bool chosenSet, chromo_t chromo);
int   measureFitness(set_t set, chromo_t chromo);
int   setDifference(set_t set, chromo_t chromo);

// Utility Functions - Program agnostic
int   randInt(int lowerBound, int upperBound);

```

```
#endif
```

File 2: Partition Problem constants and prototypes

```
/* file     : partitionProblem.c
 * author   : Channa Dias Perera (c.dias.perera@student.rug.nl)
 *          : Ola Dybvadskog     (o.dybvadskog@student.rug.nl)
 * date     : October 2020
 * version  : 1.0
 */

/* Description:
 * Functions to be used for the partition problem.
 */
#include "partitionProblem.h"
#include "simconfig.h"
#include "output.h"

// Assignment 1: Gathers the initial set containing all the numbers.
void getInitialSet(set_t set, bool autoCreateSet) {
  if (autoCreateSet) {
    for (int i = 0; i < SIZE_ORIGINAL_SET; i++) {
      /* Since we are looking for trends, as long as we have a big enough range
       * of numbers, we limit the numbers to a certain size.
       *
       * The limit on such a size would be (INT_MAX / 20)
       */

      set[i] = rand() % AUTO_GEN_SEQ_MEMBER_MAX;
    }
  } else {
    int spaceRemaining = INT_MAX;
    for (int i = 0; i < SIZE_ORIGINAL_SET; i++) {
      scanf("%d", set + i);
      if (set[i] <= 0) {
        printf("You can only have positive integers in the sequence\n");
        exit(-1);
      }

      if (set[i] > spaceRemaining) {
        printf("The input values are too large for this program to handle\n");
        printf("Make sure that the sum of the inputs is <= %d\n", INT_MAX);
        exit(-1);
      } else {
        spaceRemaining -= set[i];
      }
    }
  }
}

/* Assignment 7: Simulates evolution of the chromosomes till a solution is found
 * or we have gone many iterations without improvement.
 */
int simulateEvolution(set_t set, int *solDifference) {
  /* The total number of iterations, the number of iterations without
   * improvement.
   */
  int numIter = 0;
  int numIterNoImprov = 0;

  int convergeStatus = CONVERGING;
  chromo_t solChromo;
```

```
59
60    chromo_t generation[POP_SIZE];
61    makeInitialGenration(set, generation);
62
63    chromoCrossOver(set, generation, generation+1);
64
65    int prevBestFitness = WORST_FITNESS;
66
67    while (convergeStatus == CONVERGING) {
68      numIter++;
69      performSelection(set, generation);
70      // Check if we are converging to or have found a solution
71      convergeStatus = converges(
72        set,
73        generation,
74        &solChromo,
75        &numIterNoImprov,
76        prevBestFitness
77      );
78
79      if (numIter > MAX_ITER) {
80        convergeStatus = PAST_MAX_ITER;
81      }
82
83      if (convergeStatus != CONVERGING) {
84        // Print output
85        printOutput(set, convergeStatus, solChromo);
86        *solDifference = setDifference(set, solChromo);
87        return numIter;
88      } else {
89        // Update previous best fitness, beforem making the next generation
90        prevBestFitness = generation[BEST_CHROMO].fitness;
91        generateNewGeneration(set, generation);
92      }
93    }
94
95    return numIter;
96  }
97
98  // Makes the initial generation of chromosomes.
99  void makeInitialGenration(set_t set, chromo_t *generation) {
100   for (int i = 0; i < POP_SIZE; i++){
101     generateRandomChromo(set, &generation[i]);
102   }
103   return;
104 }
105
106 /* Performs selection. Basically replaces the weakest two chromosomes with the
107  * strongest chromosomes.
108  */
109 void performSelection(set_t set, chromo_t *generation) {
110   /* Find the weakest and strongest chromosomes. The most extreme chromosomes
111    * on either end appear earlier in the list.
112    */
113   int weakChromos[simConfig.numChromosReplaced];
114   int strongChromos[simConfig.numChromosReplaced];
115
116   /* Since we maintain the invariant that the generation would be sorted along
117    * fitness, with the fittest at the front.
118    */
119   for (int i = 0; i < simConfig.numChromosReplaced; i++) {
120     weakChromos[i]  = WORST_CHROMO-i;
```

```
    strongChromos[i] = i;
  }

  replaceChromos(
    strongChromos,
    weakChromos,
    generation,
    simConfig.numChromosReplaced
  );

  // We modified our population, so we need to sort it.
  sortChromos(generation);
}

// Check if our current generation has converged to a solution
int converges(
  set_t set,
  chromo_t *generation,
  chromo_t *solChromo,
  int *numIterNoImprov,
  int prevBestFitness
) {

  // Check if best chromosome has made improvement. Best chromo = firt chromo
  if (generation[BEST_CHROMO].fitness < prevBestFitness) {
    *numIterNoImprov = 0;
  } else {
    (*numIterNoImprov)++;
  }

  if (generation[BEST_CHROMO].fitness == 0) {
    copyChromo(solChromo, generation[BEST_CHROMO]);
    return SOLUTION_FOUND;
  } else if (*numIterNoImprov > simConfig.t2) {
    copyChromo(solChromo, generation[BEST_CHROMO]);
    return NO_IMPROVEMENT;
  }

  // No solution
  return CONVERGING;
}

/* Randomly mutates previous generation. Makes sure best chromosome is unchanged
 * Makes sure the resulting generation is sorted as well.
 * Note that the multiple mutations can occur on the same chromosome. Sometimes
 * this would result in an original mutation being reverted. This is intended.
 */
void  generateNewGeneration(set_t set, chromo_t *generation) {
  // Perform cross-overs, i.e: sexual reproduction.
  for(int i = 0; i < simConfig.numCrossOvers; i++){
    int mutator1;
    int mutator2;

    /* Determine which chromosomes to cross over. Do not cross over best one
     * which starts at index 0, so we will generate a random number greater than
     * 0.
     *
     * Equally fit solutions might exist at these indexes, but we only need to
     * preserve the best one.
     */
    mutator1 = randInt(1, POP_SIZE);
    mutator2 = randInt(1, POP_SIZE);
```

```
183
184     chromoCrossOver(set, &generation[mutator1], &generation[mutator2]);
185   }
186
187   // Perform random mutations.
188   for(int i = 0; i < simConfig.numMutations; i++){
189     // As before, avoid mutating best chromosome.
190     int mutator = randInt(1, POP_SIZE);
191
192     mutateSingleGene(set, &generation[mutator]);
193   }
194
195   // Sort this generation by fitness
196   sortChromos(generation);
197 }
198
199 /* Sort chromosomes by fitness, with the fittest occuring earlier in the aray.
200  * Use selection sort since the array is so small that O(n log n) isn't that
201  * useful.
202  */
203 void sortChromos(chromo_t *generation) {
204   for (int i = 0; i < POP_SIZE; i++) {
205     int min = i;
206     for (int j = i; j < POP_SIZE; j++) {
207       // Find smallest element
208       if (generation[j].fitness < generation[min].fitness) {
209         min = j;
210       }
211     }
212     swap(i, min, generation);
213   }
214 }
215
216 // Swap two chromosomes in the generation pool.
217 void swap(int i, int j, chromo_t *generation) {
218   chromo_t temp;
219   copyChromo(&temp, generation[i]);
220   copyChromo(generation+i, generation[j]);
221   copyChromo(generation+j, temp);
222 }
223
224 // Copy chromosome src to dst.
225 void copyChromo(chromo_t *dst, chromo_t src) {
226   for (int i = 0; i < CHROMO_LENGTH; i++) {
227     dst->genes[i] = src.genes[i];
228   }
229   dst->fitness = src.fitness;
230 }
231
232 // Replace two chromosomes, indicated by their index.
233 void replaceChromos(
234   int *strongChromos,
235   int *weakChromos,
236   chromo_t *generation,
237   int numReplaced
238 ) {
239   for (int i = 0; i < numReplaced; i++) {
240     copyChromo(generation + weakChromos[i], generation[strongChromos[i]]);
241   }
242 }
243
244 // Assignment 2: Generates a random chromosome.
```

```
245 void generateRandomChromo(set_t set, chromo_t *chromo) {
246   for (int i = 0; i < CHROMO_LENGTH; i++) {
247     chromo->genes[i] = randInt(0, 2);
248   }
249 
250   // Calculate fitness
251   chromo->fitness = 0;
252   chromo->fitness = measureFitness(set, *chromo);
253 }
254 
255 // Assignment 3: Mutate a single gene in a chromosome.
256 void mutateSingleGene(set_t set, chromo_t *chromo) {
257   int geneToMutate = randInt(0, CHROMO_LENGTH);
258 
259   chromo->genes[geneToMutate] = (chromo->genes[geneToMutate] + 1) % 2;
260   chromo->fitness = measureFitness(set, *chromo);
261 }
262 
263 // Assignment 3: Perform a cross-over muation between two chromosomes.
264 void chromoCrossOver(set_t set, chromo_t *chromo1, chromo_t *chromo2) {
265   // There is a potential for simple gene swaps, as crossOverLocation could be 0
266   int crossOverLocation = randInt(0, CHROMO_LENGTH);
267 
268   // Put the tail of chromosome1 into temp, as it will be replaced first
269   chromo_t tempChromo;
270   for (int i = crossOverLocation; i < CHROMO_LENGTH; i++) {
271     tempChromo.genes[i] = chromo1->genes[i];
272     chromo1->genes[i] = chromo2->genes[i];
273     chromo2->genes[i] = tempChromo.genes[i];
274   }
275 
276   chromo1->fitness = measureFitness(set, *chromo1);
277   chromo2->fitness = measureFitness(set, *chromo2);
278 
279 }
280 
281 /* Assignment 4: We are using the metaphor of "height" to mean the sum of one
282  * the disjoint sets.
283  *
284  * We create a set as such. For a given chromosome, each gene corresponds to
285  * a single integer in the original set. Then, the two disjoint sets are
286  * constructed as such:
287  * Set 1: All the integers that correspond to "true" genes in the chromosome
288  * Set 2: All the integers that correspond to "false" genes in the chromosome
289  */
290 int heightOfSet(set_t set, bool chosenSet, chromo_t chromo) {
291   int height = 0;
292   for (int i = 0; i < CHROMO_LENGTH; i++) {
293     if (chromo.genes[i] == chosenSet) {
294       height += set[i];
295     }
296   }
297   return height;
298 }
299 
300 /* Assignment 6: Generally, we want to have it such that our fitness increases
301  * as our difference approaches zero. However, this isn't necessary, since
302  * there's a strictly inverse relationship.
303  *
304  * Thus, if we keep this in mind, we simply need to flip our inequalities and
305  * remember that the stronger chromosomes occur earlier in a list when sorted.
306  *
```

```
 * Really, it's better this way since we don't have to process imprecise floats,
 * which we result from the inverse relationship. So while we _can_ write a
 * function, we chose not to.
 */
int measureFitness(set_t set, chromo_t chromo) {
  return setDifference(set, chromo);
}

/* Assignment 5: Instead of using heighOfSet twice, we can simply add the
 * differences as we go through the set once. Then, we take the absolute value
 * of this accumulated difference .
 *
 * Thus, instead of taking 2 * CHROMOSOME_LENGTH iterations, we can do it in
 * one iteration. Since O(n) however.
 */
int setDifference(set_t set, chromo_t chromo) {
  int diff = 0;
  for (int i = 0; i < CHROMO_LENGTH; i++) {
    /* We are finding the difference with respect to the true set. That is:
     * sum(trueSet) + diff = sum(falseSet)
     *
     */
    diff += (chromo.genes[i] ? -1 : 1) * (set[i]);
  }

  return abs(diff);
}


// Generate random int between (incl) lower and (non-incl) upper bounds.
int randInt(int lowerBound, int upperBound) {
  int r = rand();

  int range = upperBound - lowerBound;
  // Can use simple modulus technique
  if (range >= 10) {
    return lowerBound  + (r % range);
  } else {
    // Need to do it baesd on distribution
    for (int i = 0; i < range; i++) {
      if (r >= i*(RAND_MAX / range) && r < (i+1)*(RAND_MAX)/range) {
        return lowerBound + i;
      }
    }
    return lowerBound + range-1;
  }
}
```

File 3: Partition Problem functions

```
/* file     : output.h
 * author   : Channa Dias Perera (c.dias.perera@student.rug.nl)
 *          : Ola Dybvadskog     (o.dybvadskog@student.rug.nl)
 * date     : October 2020
 * version  : 1.0
 */

/* Description:
 * Function declarations for writing to stdin.
 */
#ifndef OUTPUT_H
#define OUTPUT_H
```

```

#include "partitionProblem.h"
#include "simconfig.h"

void printOriginalSet(set_t set);

void printOutput(set_t set, int convergeStatus, chromo_t chromo);

void printChromo(chromo_t chromo);

void printSets(set_t set, chromo_t chromo);
void printSet(set_t set, bool chosenSet, chromo_t chromo);

void printDivider(int len);

#endif
```

File 4: Constants and prototypes related to output

```
/* file     : output.c
 * author   : Channa Dias Perera (c.dias.perera@student.rug.nl)
 *          : Ola Dybvadskog     (o.dybvadskog@student.rug.nl)
 * date     : October 2020
 * version  : 1.0
 */

/* Description:
 * Functions definitions to be used for writing to stdin.
 */
#include "output.h"
#include "partitionProblem.h"
#include "simconfig.h"
#include "testing.h"

// A debugging function, that prints the original set.
void printOriginalSet(set_t set) {
  for (int i = 0; i < SIZE_ORIGINAL_SET; i++) {
    printf("%d ", set[i]);
  }
  printf("\n");
}

// Prints the output for the program based on the solution.
void printOutput(set_t set, int convergeStatus, chromo_t chromo) {
#if !TESTING
  // Print type of solution
  switch(convergeStatus) {
    case SOLUTION_FOUND:
      printf("ZERO'D difference of sums!\n");
      break;
    case PAST_MAX_ITER:
    case NO_IMPROVEMENT:
      printf(
        "MINIMIZED absolute difference of sums: %d!\n",
        setDifference(set, chromo)
      );
      break;
    default:
      break;
  }

  // Print chromosome
```

```
  printf("CHROMOSOME: ");
  printChromo(chromo);
  printDivider(DIVIDER_LEN);

  // Print set results
  printSets(set, chromo);
  printDivider(DIVIDER_LEN);
#endif
}

// Prints a specific chromosome to stdin.
void printChromo(chromo_t chromo) {
  for(int i = 0; i < CHROMO_LENGTH; i++) {
    printf("%d ", chromo.genes[i]);
  }
  printf("\n");
}

// Prints out the sets from a chromosome
void printSets(set_t set, chromo_t chromo) {
  // Print true set
  printf("COLLECTION 1: ");
  printSet(set, true, chromo);


  // Print false set
  printf("COLLECTION 2: ");
  printSet(set, false, chromo);
}

// Prints out a specific set from a chromosome
void printSet(set_t set, bool chosenSet, chromo_t chromo) {
  bool prevNumberPresent = false;

  int sum = 0;

  printf("{");
  for (int i = 0; i < CHROMO_LENGTH; i++) {
    if (chromo.genes[i] == chosenSet) {
      // Add the preceding comma, if there was a number before the current num
      if (prevNumberPresent) {
        printf(", ");
      }
      prevNumberPresent = true;
      printf("%d", set[i]);

      sum += set[i];
    }
  }
  printf("}\n");

  printf("sum = %d", sum);
  printf("\n");
}

void printDivider(int len) {
  for (int i = 0; i < len; i++) {
    printf("%c", DIVIDER_CHAR);
  }
  printf("\n");
```

```
}
```

File 5: Functions related to output

```
/* file     : simConfig.h
 * author   : Channa Dias Perera (c.dias.perera@student.rug.nl)
 *          : Ola Dybvadskog     (o.dybvadskog@student.rug.nl)
 * date     : October 2020
 * version  : 1.0
 */

/* Description:
 * Constants, types and function declarations relation to the simulation
 * configuration.
 */
#ifndef SIMCONFIG_H
#define SIMCONFIG_H

#include <stdio.h>
#include <stdbool.h>

#define  SIZE_ORIGINAL_SET 20
#define CHROMO_LENGTH SIZE_ORIGINAL_SET

// Termination conditions
#define MAX_T2 500000
#define MAX_ITER (10 * MAX_T2)

#define POP_SIZE 10

// Program configuration
#define NUM_CROSS_OVERS 4
#define NUM_MUTATIONS 20
// Maximum = POP_SIZE / 2
#define NUM_CHROMOS_REPLACED 2

/* Global simulation configuration. Since it's related to all the functions in
 * we decided to make it a global function.
 *
 * If we decide to test multiple configurations programatically, this struct can
 * be used.
 */
typedef struct {
  int numCrossOvers;
  int numMutations;
  int numChromosReplaced;
  int t2;
} simConfig_t;

simConfig_t simConfig;

// FUNCTIONS

void printConfig(simConfig_t simConfig);

#endif
```

File 6: Constants and prototypes related to simulation configuration

```
/* file     : simconfig.c
 * author   : Channa Dias Perera (c.dias.perera@student.rug.nl)
 *          : Ola Dybvadskog     (o.dybvadskog@student.rug.nl)
```

```
 * date     : October 2020
 * version  : 1.0
 */

/* Description:
 * Function relation to the simulation configuration.
 */
#include "simconfig.h"

// Prints out the coniguration information
void printConfig(simConfig_t simConfig) {
  printf("Simulation Configuration:\n");
  printf(
    "Cross-overs: %d, Mutations: %d, Selection Replacement: %d, t2: %d\n\n",
    simConfig.numCrossOvers,
    simConfig.numMutations,
    simConfig.numChromosReplaced,
    simConfig.t2
  );
}
```

File 7: Functions related to simulation configuration

```
/* file     : testing.h
 * author   : Channa Dias Perera (c.dias.perera@student.rug.nl)
 *          : Ola Dybvadskog     (o.dybvadskog@student.rug.nl)
 * date     : October 2020
 * version  : 1.0
 */

/* Description:
 * Constants, types and function declarations relation to testing the program
 */
#ifndef TESTING_H
#define TESTING_H

#include <stdio.h>
#include <stdbool.h>
#include <limits.h>

#include "simconfig.h"
#include "partitionProblem.h"

// Constants

// Setting of program. If we're testing and what sort of testing.
#define TESTING 0

// How finely grained should T2 be
#define T2_GRAIN 10
// The minimum percentage similarity sougth
#define T2_MIN_PER_SIM 0.8

#define T2_CONFIG_NUM_SIMS 100

// Functions
void findT2();
bool sufficientT2(int t2);
int   findAverageIterationLength(int *longestIter);

#endif
```

File 8: Constants and prototypes related to testing

```c
/* file     : testing.c
 * author   : Channa Dias Perera (c.dias.perera@student.rug.nl)
 *          : Ola Dybvadskog     (o.dybvadskog@student.rug.nl)
 * date     : October 2020
 * version  : 1.0
 */

/* Description:
 * Function relation to the testing the program.
 */

#include "testing.h"
#include "string.h"
#include "output.h"

void findT2() {
  simConfig.numChromosReplaced = NUM_CHROMOS_REPLACED;
  simConfig.numCrossOvers = NUM_CROSS_OVERS;
  simConfig.numMutations = NUM_MUTATIONS;
  int t2 = 1;
  while (!sufficientT2(t2)) {
    t2 *= 10;
  }

  // Fine grain t2
  int fineGrainedT2;
  for (int i = 1; i <= T2_GRAIN; i++) {
    if (sufficientT2((i * t2) / T2_GRAIN)) {
      fineGrainedT2 = ((i * t2) / T2_GRAIN);
    }
  }

  printf("Use t_2 of %d\n", fineGrainedT2);

}

bool sufficientT2(int t2) {
  printf("t_2 = %d\n", t2);

  simConfig.t2 = t2;
  int results[T2_CONFIG_NUM_SIMS];
  set_t set;
  getInitialSet(set, true);
  // Fill results
  for (int i = 0; i < T2_CONFIG_NUM_SIMS; i++) {
    // See how far through this run of t_2 we are
    if (i % (T2_CONFIG_NUM_SIMS / 50) == 0) {
      printf("|");
      fflush(stdout);
    }

    simulateEvolution(set , results + i);
  }
  printf("|\n");

  // See how well this t2 did
  int smallestVal = results[0];
  int numSmallest = 1;
  for (int i = 1; i < T2_CONFIG_NUM_SIMS; i++) {
    if (smallestVal == results[i]) {
      numSmallest++;
    } else if (smallestVal > results[i]) {
```

```
      smallestVal = results[i];
      numSmallest = 1;
    } else {
      continue;
    }
  }
  printf(
    "Percent similarity: %d%%\n\n",
    (int) (numSmallest * 100.0) / T2_CONFIG_NUM_SIMS
  );

  // Check if t2 is good enough. Floating point math here, but that's okay
  if (numSmallest >= T2_MIN_PER_SIM * T2_CONFIG_NUM_SIMS) {
    return true;
  } else {
    return false;
  }

}
```

File 9: Functions related to testing

File 10: The makefile for the program

```
fileName = partitionProblem

$(fileName): $(fileName).o simconfig.o output.o testing.o main.c makefile
  gcc main.c $(fileName).o simconfig.o output.o testing.o -o $(fileName).exe

$(fileName).o: $(fileName).c $(fileName).h makefile $(wildcard *.h)
  gcc -std=c99 -g -Wall -pedantic -c $(fileName).c -o $(fileName).o -I.

simconfig.o: simconfig.c $(wildcard *.h) makefile
  gcc -std=c99 -g -Wall -pedantic -c simconfig.c -o simconfig.o -I.

output.o: output.c $(wildcard *.h) makefile
  gcc -std=c99 -g -Wall -pedantic -c output.c -o output.o -I.

testing.o: testing.c $(wildcard *.h) makefile
  gcc -std=c99 -g -Wall -pedantic -c testing.c -o testing.o -I.

test: $(fileName)
  ./$(fileName).exe < tcADS0.in
  echo
  ./$(fileName).exe < tcADSn0.in
  echo
  ./$(fileName).exe < tcADSLarge.in
  echo
  ./$(fileName).exe < tcSameNumber.in
  echo
  ./$(fileName).exe < tcRandom.in

run: $(fileName)
  ./$(fileName).exe

grabTrends: $(fileName)
  ./$(fileName).exe > ../results/trends.txt
```

```
clean:
  rm -f *.o $(fileName).exe
```