A01

Merkle–Hellman knapsack cryptosystem

Arquitetura e Estruturas de Dados

Universidade Aveiro

(103320) Bruno Gomes - 40%

(103552) Alexandre Martins - 30%

(104090) Tomás Rodrigues - 30%

6 de Janeiro de 2022

Conteúdo

Capítulo 1

Introdução

Merkle–Hellman é um criptosistema de chave pública, baseado no ***Subset sum problem*** (um problema de soma de subconjuntos) um caso particular, do ***Knapsack problem*** (problema da mochila).

O problema é o seguinte: dado um conjunto de valores inteiros **A** e um valor inteiro **c**, encontre um subconjunto de **A** que soma **c**. Em geral, este problema é conhecido como ***NP*** - completo (***Knapsack*** - completo). No entanto, se **A** está a aumentar exponencialmente, o que significa que cada elemento do conjunto é maior do que a soma de todos os números no conjunto menor do que ele, o problema é "fácil"e solucionável em tempo polinomial com um simples algoritmo "***greedy***".

Neste projeto, A01, foi nos solicitada a resolução de pequenos segmento deste problema:

- **Força bruta**;
- Força bruta inteligente (chamada de ***branch and bound***);
- Técnica de Horowitz e Sahni (técnica de ***meet-in-the-middle***).

Na realização da Técnica de Horowitz e Sahni (técnica de ***meet-in-the-middle***, utilizamos algumas ferramentas da internet:

- https://www.geeksforgeeks.org/power-set/
- https://www.geeksforgeeks.org/selection-sort/
- https://technotip.com/8920/c-program-to-divide-split-an-array-into-two-at-specified-position/

Capítulo 2

Metodologia

Um dos métodos em questão seriam ***brute force*** que consiste simplesmente em enumerar todas as possibilidades procurando entre estas candidatos que satisfazem o requisitos de modo a se qualificarem como uma resolução para o problema.

O método ***branch and bound*** é uma abordagem de solução que divide o espaço de solução viável em subconjuntos menores de soluções. Consiste também em uma enumeração sistemática de todos os candidatos a solução, através da qual os subconjuntos de candidatos infrutíferos são descartados em massa utilizando os limites superior e inferior da quantia otimizada.

Por fim temos o método de Horowitz e Sahni, também conhecido como ***meet-in-the-middle*** que consiste em dividir o problema em dois, resolve-os individualmente e depois junta-os.

Tendo essa informação em conta, o objetivo do projeto foi adaptar estes métodos ao problema dado.

No Capítulo 4 segue o código completo com comentários, estes incluem os criados pelo professor e a descrição do código produzido pelos alunos.

Capítulo 3

Resultados

3.1 Comparação do desempenho das funções

- **Brute force:** Esta função é a mais lenta de todas, pois através da recursividade esta vai testando todas a combinações de somas possíveis para achar cada solução, daí os tempos serem irregulares pois existem casos em que a solução é encontrada de imediato, no entanto também existem situações em que se tem de experimentar todas as somas possíveis para encontrar a respetiva solução.
- **Branch and Bound:** Esta função apresenta tempos mais reduzidos, no entanto também usa recursão tal como o "brute force"a diferença é que nesta função estão presentes algumas condições que fazem com que a procura de soluções seja mais eficiente, como por exemplo caso a soma parcial somada com o resto dos elementos que restam somar sejam menores que a soma desejada, o programa para e devolve 0(não achou solução), adicionalmente também existe a situação em que a soma parcial é maior que a soma desejada aqui o programa também retorna o número 0 pois não vai ser necessário verificar outras somas.
- **Meet in the Middle:** Esta função é a mais rápida de todas, no entanto apresenta um aumento exponencial dos tempos de execução para cada problema, o programa podia ser mais rápido se não tívessemos usado a mesma função duas vezes (powerSet), adicionalmente a demora pode vir do malloc que é feito todas as vezes que se cria um array dentro da função. Depois do problema 33 já é bastante complicado e demoroso obter uma solução.

Todos os testes foram realizados para n>33 pois a partir desse número já era bastante demoroso obter mais soluções.

Capítulo 4

Código

```
//
// AED, November 2021
//
// Solution of the first practical assignement (subset sum problem)
//
// Place your student numbers and names here
//

#if __STDC_VERSION__ < 199901L
# error "This code must must be compiled in c99 mode or later (-std=c99)" // to handle the unsigned long long data type
#endif
#ifndef STUDENT_H_FILE
# define STUDENT_H_FILE "104090.h" // muda conforme o estudante
#endif


//
// include files
//

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "elapsed_time.h"
#include STUDENT_H_FILE


//
// custom data types
//
// the STUDENT_H_FILE defines the following constants and data types
//
//   #define min_n       24                  --- the smallest n value we will handle
//   #define max_n       57                  --- the largest n value we will handle
//   #define n_sums      20                  --- the number of sums for each n value
//   #define n_problems  (max_n - min_n + 1)  --- the number of n values
//
//   typedef unsigned long long integer_t;    ---  64-bit unsigned integer
//   typedef struct
//   {
//     int n;                                --- number of elements of the set (for a valid problem, min_n <= n <= max_n)
//     integer_t p[max_n];                   --- the elements of the set, already sorted in increasing order (only the first n elements are used)
//     integer_t sums[n_sums];               --- several sums (problem: for each sum find the corresponding subset)
//   }
//   subset_sum_problem_data_t;              --- weights p[] and sums for a given value of n
//
//   subset_sum_problem_data_t all_subset_sum_problems[n_problems]; --- // the problems
//


//
// place your code here
//
// possible function prototype for a recursive brute-force function:

//BRUTE FORCE
int brute_force(int n,integer_t p[n],integer_t desired_sum,int next_index,integer_t partial_sum,int b[n]){

  if(desired_sum==partial_sum){
    for(int i = next_index;i<n;i++){
      b[i] = 0;
```

```
    }
    return 1; //achou uma solução
  }

  if(next_index==n){
    return 0; //chegou ao fim do array e não achou solução
  }
  //set next bit to zero
  b[next_index] = 0;
  int result = brute_force(n,p,desired_sum,next_index + 1,partial_sum,b);

  if(result==1){
    return 1;
  }

  //set next bit to one
  b[next_index] = 1;

  return brute_force(n,p,desired_sum,next_index+1, partial_sum + p[next_index],b);


} // n->array size, integer_t *p -> data array, integer_t desired_sum, int next_index, integer_t partial-sum, b-> array binário
// it sould return 1 when the solution is found and 0 otherwise
// note, however, that you may get a faster function by reducing the number of function arguments (maybe a single pointer to a struct?)
//

//BRANCH AND BOUND
int branch_and_bound(int n,integer_t p[n],integer_t desired_sum,int next_index,integer_t partial_sum,int b[n]){
  //printf("next index = %d partial sum = %lld \n",next_index, partial_sum);
  //casos base
  int i,k;
  if(desired_sum==partial_sum){
    for(i = next_index;i<n;i++){
      b[i] = 0;
    }
    for(k = 0; k<n; k++){
      if(b[k] == 1){
      }
    }
    return 1;
  }

  if(next_index==n){
    return 0; //chegou ao final e não achou solução
  }

  integer_t remaining_sum = 0;

  for(i = next_index; i<n; i++){
    remaining_sum += p[i];
  }

  if((partial_sum+remaining_sum) < desired_sum){
    return 0; //ignora o resto das somas pois a soma parcial mais o resto é muito pequena
  }

  if(partial_sum>desired_sum){
    return 0; //soma parcial muito grande
  }

  //set next bit to 0
  b[next_index] = 0;
  int result = branch_and_bound(n,p,desired_sum,next_index+1,partial_sum,b);
  if(result == 1){
    return 1;
  }

  //set next bit to 1
  b[next_index] = 1;
  return branch_and_bound(n,p,desired_sum,next_index+1,partial_sum + p[next_index],b);
}

//MEET IN THE MIDDLE + funções auxiliares
int checkDesiredSum(int arrA[], int arrB[],int aLength, int bLength,int ds){


    int i = 1;
    int j = bLength-1;

    while(i < aLength && j >= 1){
        if(arrA[i] + arrB[j] == ds)
            return 1; //existe solução
        if(arrA[i] + arrB[j] < ds)
            i++;
        if(arrA[i] + arrB[j] > ds)
            j--;
    }

    return 0; //solução não existe

}
```

```
void swap(int *xp, int *yp){ //trocar posições dentro do array(essencial para o selectionSort funcionar...)
  int temp = *xp;
  *xp = *yp;
  *yp = temp;
}

void selectionSort(int arr[], int n){ //algoritmo para ordenar o array que tem as combinações das somas possíveis
  int i,j, min_idx;

  for(i = 0; i< n-1;i++){
    min_idx = i;
    for(j = i +1; j<n;j++){
      if(arr[j]<arr[min_idx]){
        min_idx = j;
      }
    }
    swap(&arr[min_idx], &arr[i]);
  }

}



int arraySizeB; //variável global para armazenar tamanho do array

int *powerSetB(int *set, int set_size) //função auxiliar para calcular todas as somas possiveis através de arrays binários
{
    /*set_size of power set of a set with set_size
      n is (2**n -1)*/
    unsigned int pow_set_size = pow(2, set_size);
    arraySizeB = pow_set_size;
    int counter, j;
    //int sums[pow_set_size-1]; // tirar o 0
    static int* sums;
    sums = malloc((pow_set_size-1)*sizeof(int));
    /*Run from counter 000..0 to 111..1*/
    for(counter = 0; counter < pow_set_size; counter++)
    {

      int sum = 0;
      for(j = 0; j < set_size; j++)
       {
          /* Check if jth bit in the counter is set
             If set then print jth element from set */
          int bit = counter & (1<<j);
          if(bit){

            sum = sum + set[j];

          }
       }

       if(sum != 0){
         sums[counter] = sum;
         //printf("sums[%d] = %d\n",counter, sums[counter]); //print para ver todas as somas possíveis
       }

       //printf("\n");
    }
    return sums;
}



int arraySizeA; //variável global para armazenar tamanho do array

int *powerSetA(int *set, int set_size) //função auxiliar para calcular todas as somas possiveis através de arrays binários
{
    /*set_size of power set of a set with set_size
      n is (2**n -1)*/
    unsigned int pow_set_size = pow(2, set_size);
    arraySizeA = pow_set_size;
    int counter, j;
    //int sums[pow_set_size-1]; // tirar o 0
    static int* sums;
    sums = malloc((pow_set_size-1)*sizeof(int));
    /*Run from counter 000..0 to 111..1*/
    for(counter = 0; counter < pow_set_size; counter++)
    {

      int sum = 0;
      for(j = 0; j < set_size; j++)
       {
          /* Check if jth bit in the counter is set
             If set then print jth element from set */
          int bit = counter & (1<<j);
          if(bit){
```

```
            sum = sum + set[j];

          }
        }

        if(sum != 0){
          sums[counter] = sum;
          //printf("sums[%d] = %d\n",counter, sums[counter]); //print para ver todas as somas possíveis
        }

        //printf("\n");
    }
    return sums;
}



int meet_in_the_middle(int n,integer_t p[n], integer_t desired_sum){

  //algoritmo para dividir arrays
  int i = 0, j = 0, pos = n/2, x[pos], y[n-pos];

  for(int k = 0; k<n ;k++){
    if(k<pos){
      x[i++] = p[k];
    }
    else{
      y[j++] = p[k];
    }
  }
  //algoritmo para dividir arrays(end)

  int length = sizeof(x)/sizeof(x[0]); //maneira de obter comprimento de arrays
  int length1 = sizeof(y)/sizeof(y[0]);


  int *a, *b;

  a = powerSetA(x, length); // -> Combinações de somas possiveis(num array) através de arrays binários (para o array x)
  b = powerSetB(y, length1); // -> Combinações de somas possiveis(num array) através de arrays binários (para o array y)



  selectionSort(a,arraySizeA);  // -> array somas de 'a' ordenado
  selectionSort(b,arraySizeB); // -> array somas de 'b' ordenado


  return checkDesiredSum(a,b,arraySizeA,arraySizeB,desired_sum); // devolve 0 se não achar solução, devolve 1 se achar solução

}

//
// main program
//

int main(void)
{

  fprintf(stderr,"Program configuration:\n");
  fprintf(stderr,"  min_n ....... %d\n",min_n);
  fprintf(stderr,"  max_n ....... %d\n",max_n);
  fprintf(stderr,"  n_sums ...... %d\n",n_sums);
  fprintf(stderr,"  n_problems .. %d\n",n_problems);
  fprintf(stderr,"  integer_t ... %d bits\n",8 * (int)sizeof(integer_t));
  //
  // for each n
  //

  for(int i = 0;i < n_problems;i++)
  {

    int n = all_subset_sum_problems[i].n; // the value of n
    if(n > 33) // n > 33 valor original
      continue; // skip large values of n
    integer_t *p = all_subset_sum_problems[i].p; // the weights
    //
    // for each sum
    //

    printf("\n");
    printf("************************************************************************\n");
    printf("\n");


    for(int j = 0;j < n_sums;j++)
    {

      integer_t desired_sum = all_subset_sum_problems[i].sums[j]; // the desired sum
      int b[n]; // array to record the solution meter n
      //
```

```
      // place your code here
      //

      //teste brute force
      /*
      double begin = cpu_time();

      printf("%d \n",brute_force(n,p,desired_sum,0,0,b));

      double end = cpu_time();
      double resultado1 = end - begin;
      printf("cputime = %f \n",resultado1);
      */
      //end teste


      //teste branch and bound
      /*
      double begin1 = cpu_time();

      printf("%d \n",branch_and_bound(n,p,desired_sum,0,0,b));

      double end1 = cpu_time();
      double resultado2 = end1 - begin1;
      printf("cputime = %f \n",resultado2);
      */

      //end teste

      //teste meet in the middle

      double begin2 = cpu_time();


      printf("%d\n",meet_in_the_middle(n,p,desired_sum));

      double end2 = cpu_time();
      double resultado3 = end2 - begin2;
      printf("cputime = %f \n",resultado3);

      //end teste


    }


  }
 return 0;

}
```