AED - Algoritmos e Estruturas de Dados
Hash Table implementation

Hash Tables implementation using singly linked lists and binary trees

Dinis Cruz	Duarte Mortágua	Tiago Oliveira
92080	92963	93456
33,3%	33,3%	33,3%

Teacher: Tomás Oliveira e SIlva

Departamento de Eletrónica,

Telecomunicações e Informática

Universidade de Aveiro

december 27, 2019

Contents

List of Tables . . . . . **iv**

List of Figures . . . . . **iv**

1 Introduction **1**

1.1 A brief description of the problem 1

2 Implementation methods **3**

2.1 Singly linked lists implementation 3

2.2 Binary trees implementation 5

3 Dynamic resizing **7**

3.1 Approach description 7

3.1.1 Singly linked lists 8

3.1.2 Binary trees 9

4 Tests **11**

4.1 Singly linked lists tests 11

4.1.1 Output 11

4.1.2 Execution time analysis 12

4.2 Binary trees solution 13

4.2.1 Output 13

4.2.2 Execution time analysis 15

4.3 Dynamic resizing 16

4.4 Solution comparison 16

A Code appendix **20**

Bibliography **20**

List of Tables

List of Figures

3.1 Adding word1 to empty linked list 8

3.2 Adding word2 to linked list containing word1 9

4.1 Distribution for $n = 14$ 12

4.2 Execution time - Brute force with branch-and-bound method 15

4.3 Execution time - Comparison of both solutions 18

1. Introduction

1.1 A brief description of the problem

To start of, the main goal of this practical work was the implementation of an hash-table. The purpose of an hash-table implementation is to store data, so we can see an hash-table as a data structure. With the hast-table implementation the main goal was to store all the different words in a determined text and how many times every single word occurred in that text in each linked list entry corresponding each hash-table entry. Besides, we were told to store in each linked-list entry the first and last locations of the word and the maximum, minimum and medium distances between words. As we had to store a lot of information about each distended word, we thought that was clever to use structures as the representation of each word and the linked lists entries would be this structures. The language we used to implement this was C, although we are not very familiar with it we think we have done a good job using it. As soon as we started thinking about the implementation we decided that the data that we were going to store would pass through an hash function first, however as we all know hashing methods always have collisions and as we want to have a efficient implementation we decided to use, in every hash-table entry, a linked list. Liked lists are basically separated elements that have a pointer to next element and so on, in this way every time we got an hashing collision we would put the next element in the linked list and the element that was already there with a pointer to it. As soon as we finish implementing the hash-table with linked lists we thought we could implement the hash-table but now with binary trees instead of linked lists to compare which one is the most efficient way.

2. Implementation methods

2.1 Singly linked lists implementation

As we described earlier, the linked list implementation is based in each element pointing to the next. Every time we have an hashing collision the linked list in that hashing position will grow. Nevertheless, every time we hash something to the hash-table we need to check if that position is still empty or not and if the word we are hashing is already in the hash-table or not. First of all, as soon as we have the hashing position that the word we are hashing will occupy, we go through the linked list in that position and in each element we check if it matches the word we are storing. If we find a match then we update all the data of that structure(which are what we used to represent each word) such as the number of times it occurred and the last, maximum, minimum and medium distances. On the other hand, while we are going trough the linked list if we reach a pointer that points do NULL then we know for sure that we are checking the last element of the linked list and as we have not got any match this means that the word we are hashing still does not exists in the hash-table so we need to add him. As we already described the adding in the linked list is basically changing the pointer of the last element of the linked list to point to the new element so the new element becomes the last element of the that linked list.

2.2 Binary trees implementation

The binary trees implementation is based in having more efficiency in the search. Different than the linked list implementation, every structure in the hash table has a pointer to a binary tree. We thought that it would not make sense if we used unordered binary trees because that's what makes the search efficient. Of course every binary tree node has the same information as the linked list structure, such as the number of times the word occurred and the last, minimum, maximum and medium distances and a few more details we found imperative. On one hand, the collisions problem is solved exactly the same way the linked list implementation is, every time we have a collision, if the hashing position has already a pointer to the head of the binary tree, we go through all the binary tree and once we arrive the end of the tree if we found no match to the word then we add a new node to the binary tree.

On the other hand, the adding of a new node is different. As the binary tree is ordered, every time we have to add a new node to a tree, we compare the word we are adding with the head, if it is lower we go left, if it is greater we go right, and we go through the binary tree until we find the exact spot that fits the word.

About the resize, we did not implement it a long with the binary tree implementation because the efficiency is in the capacity of the search being shorter, and so we think that a resize would not be a must need functionality (see 3.1.2).

3. Dynamic resizing

In order to reduce the number of collisions as the hash table gets filled, we implemented a dynamic resize function that doubles the size of the hash table and reallocates the words in the table. This action is triggered whenever the load factor is reached. The load factor is the number of linked lists (or binary trees) stored in the hash table divided by it’s capacity, and in this case, we use a load factor of 0.5.

3.1 Approach description

Our hash table structure is not only characterized by the table itself, but by the elements "count" and "size" too. Whenever a word is read and occupies a new entry of the hash table, the count element is increased, so the count keeps track of the hash table slots that are not null. The size initially represents the number of slots of the hash table, equaling 2000. When $count/size >= 0.5$, the resize action is triggered.

3.1.1 Singly linked lists

We start the resize process by creating a new empty table of pointers to words (our new hash table). This new table will have the size equal to two times the old size. Then, we iterate thought the hash table entries. In each one, we detach the singly link list, keeping track of the head pointer in a new variable, "next". Then, as the size will be changed and so the hash-code too, we cant just move the entire linked list to a new entry of the new table, we have to detach the entire linked list and map each one of it’s words to a new linked list in the new hash table. In order to do that, and having the "next" pointer, we iterate through all the words of each linked list, and since each word has an attribute "hash" (djb2 direct hash), we get the index of the word’s new entry in the new hash table by getting the rest of the division of the hash by the size of the new hash table. After getting the index, instead of appending the word to the end of the new linked list, we attach it to the beginning. We do this by causing the new word to point to the first word in the list, and then replace the first word in the list with the new word.

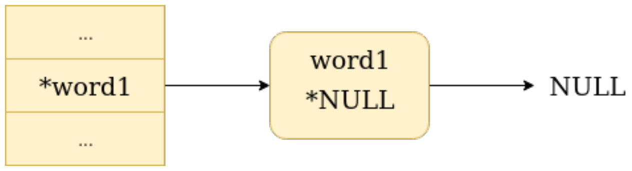

Figure 3.1: Adding word1 to empty linked list

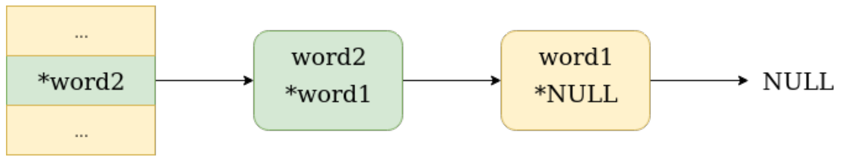

Figure 3.2: Adding word2 to linked list containing word1

After doing this to all the linked lists in the old hash table, we free the memory of the old hash table, replace it by the new one and update the hash table size with the new size.

3.1.2 Binary trees

We decided not to implement dynamic resize when implementing the hash table with binary trees because:

- When talking about adding elements, resizing the table or not, the time complexity is $O(1)$ for both implementations, being that the dynamic resize would only have impact in time execution when searching for a word in the table.
- Each time we do the dynamic resize, in linked lists, we go through all the words in all the old lists one by one ($O(n)$) and add them to the new linked lists in the new table ($O(1)$), having a total complexity of ($O(n)$). On the other hand, in binary trees, we go through all the words in all the old trees one by one ($O(n)$) and then insert them on the correct position in the new binary trees ($O(n)$), having a total complexity of ($O(n^2)$). So, resizing when the

objective is adding words to the hash table isn't such a great idea when talking about binary trees implementation.

- Ignoring dynamic resize in binary trees doesn't affect the complexity that much when searching, because the fact of using ordered binary trees, although huge, decreases the search time of O(n) of the linked lists search to $O(logn)$, so it becomes fast enough to keep up with the search when dealing with linked lists (resized).

In sum, since the proposed problem was to add words to the hash table, we thought it would be less time expensive to implement an hash table with singly linked lists with dynamic resizing and an hash table with binary trees without dynamic resizing.

4. Tests

4.1 Singly linked lists tests

4.1.1 Output

Using brute force method we were able to compute a solution for the assignment problem for $n \leqslant 14$. Below is presented the output of the solution, containing the execution time (cpu time), costs matrix generated, minimum cost found and respective assignment and maximum cost found and respective assignment, for each n.

4.1.2 Execution time analysis

We have concluded that the execution time increases exponentially as n increases. It's quite expected, since it's given an n-by-n costs matrix, $n!$ permutations need to be calculated in order to get the *min_cost* and *max_cost* permutation out of the permutations set.

The graphic above was made using the solution output, concluding the exponential growth of the execution time.

`figures/n14.jpg`

Figure 4.1: Distribution for $n = 14$

The graphics above were made using the solution output, concluding that as n increases, the number of occurrences of the permutations cost's follows a normal distribution.

As we can see from the above graphic, as n increases, the average of the costs tends to move to the right in a constant rhythm. This can be explained because we have a fixed minimum cost for the permutations of each n, such as $min\_cost = 3 * n$, and a fixed maximum cost such as $max\_cost = 60 * n$. Our Gaussians are centred between this intervals, so for $n = 14$ the peak of the Gaussian will be near $(min\_cost + max\_cost)/2$ or $(3 * 14 + 60 * 14)/2 = 441$. Similarly for $n = 13$ the peak will be near 410 and for $n = 12$ near 380. This values are spaced of, approximately, 30, which is a constant.

We can see that the maximum occurrences for each n increase exponentially. This is due the fact that for each n we have approximately $n!$ permutations, so each cost occurrence tends to increase towards that growth rhythm.

4.2 Binary trees solution

4.2.1 Output

Using brute force with branch-and-bound method we were able to compute a solution for the assignment problem for $n \leqslant 16$. Below is presented the output of the solution, containing the execution time (CPU time), costs matrix generated, minimum cost found and respective assignment and maximum cost found and respective assignment, for each n.

4.2.2 Execution time analysis

It is expected that the execution time increases exponentially as n increases, but at a lower speed than in the brute force solution without branch-and-bound, since as explained in the branch-and-bound method, the number of permutations calculated is lower than $n!$.

figures/cpu_time_bab.jpg

Figure 4.2: Execution time - Brute force with branch-and-bound method

The graphic above was made using the solution output, concluding the exponential growth of the execution time.

4.3 Dynamic resizing

This method is the only method that is simulation based (never 100% accurate).
Due to that, in terms of method comparisons, in this section of the report we will simply show the output and make a comparison between the minimum costs found by the Random Permutations method and the Branch-and-bound method.
We are also going to analyse the approximation of the obtained maximum and minimum costs in this method to the theoretical maximum and minimum costs.

4.4 Solution comparison

The use of the Random Permutations method rather than the other two methods is justifiable and reliable for certain values of n (as mentioned), as it is a simpler and relatively fast algorithm.

But as n increases, this simulation based method tends to be inappropriate and much less accurate.

Both Brute Force method and Brute Force with branch-and-bound solutions are brute force based solutions, but the branch-and-bound solution is significantly faster for higher values of n. This is due to the fact that brute force without branch-and-bound has to inevitably calculate $n!$ permutations with n costs to sum each, giving us the "hardest" and less efficient method for the assignment problem.

In the other hand, brute force with branch-and-bound algorithm skips the calculation of some permutations the moment it knows that they will inevitably become more costly than the best permutation already found so far, avoiding a great number of calculations and therefore reducing the total execution time.

figures/cpu_time_comparison.jpg

Figure 4.3: Execution time - Comparison of both solutions

The graphic above was made using the solution output, confirming the exponential growth of the execution time of both solutions and the lowest growth rate of the brute force solution with branch-and-bound. The branch-and-bound based solution represents a notorious increase of performance, since for $n = 14$ the difference of execution times is about

10^1 times, and making a speculation based on the rhythm of growth of both curves, as n increases, the difference of execution times increases at a constant rhythm too.

Concluding, the branch-and-bound solution becomes much more efficient as n increases, compared to the solution without branch-and-bound or the Random Permutations solution.

A. Code appendix

Below is the code of our assignment.c file. All the comments made by us are in capital letter. Some comments made by the teacher were omitted in order to keep this appendix concise. These comments were no longer relevant since they were supposed to guide our code work in an early stage.

```c
////////////////////////////////////////////////////////////////////////////////////////////////////
//
// AED, 2019/2020
//
// DINIS CRUZ 92080
// DUARTE MORT GUA 92963
// TIAGO OLIVEIRA 93456
//
// Brute-force solution of the assignment problem
// (https://en.wikipedia.org/wiki/Assignment_problem)
//
// Compile with "cc -Wall -O2 assignment.c -lm" or equivalent
//
// In the assignment problem we will solve here we have n agents and n tasks;
// assigning agent
//   a
// to task
//   t
// costs
//   cost[a][t]
// The goal of the problem is to assign one agent to each task such that the
// total cost is minimized
// The total cost is the sum of the costs

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

//#define NDEBUG  // uncomment to skip disable asserts (makes the code slightly
// faster)
#include <assert.h>

////////////////////////////////////////////////////////////////////////////////////////////////////
//
// problem data
//
// max_n ........ maximum problem size
// cost[a][t] ... cost of assigning agent a to task t
//
//
// if your compiler complains about srandom() and random(), replace #if 0 by #if
// 1
//
#if 0
```

```c
#define srandom srand
#define random rand
#endif

#define max_n 32 // do not change this (maximum number of agents, and tasks)
#define range                                                                  \
  20 // do not change this (for the pseudo-random generation of costs)
#define t_range                                                                \
  (3 * range) // do not change this (maximum cost of an assignment)

static int cost[max_n][max_n];
static int seed; // place a student number here!

static void init_costs(int n) {
  if (n == -3) { // special case (example for n=3)

    cost[0][0] = 3;   cost[0][1] = 8;   cost[0][2] = 6;
    cost[1][0] = 4;   cost[1][1] = 7;   cost[1][2] = 5;
    cost[2][0] = 5;   cost[2][1] = 7;   cost[2][2] = 5;

    return;
  }
  if (n == -5) { // special case (example for n=5)

    cost[0][0] = 27;    cost[0][1] = 27;    cost[0][2] = 25;    cost[0][3] = 41;    cost[0][4] = 24;
    cost[1][0] = 28;    cost[1][1] = 26;    cost[1][2] = 47;    cost[1][3] = 38;    cost[1][4] = 21;
    cost[2][0] = 22;    cost[2][1] = 48;    cost[2][2] = 26;    cost[2][3] = 14;    cost[2][4] = 24;
    cost[3][0] = 32;    cost[3][1] = 31;    cost[3][2] = 9;     cost[3][3] = 41;    cost[3][4] = 36;
    cost[4][0] = 24;    cost[4][1] = 34;    cost[4][2] = 30;    cost[4][3] = 35;    cost[4][4] = 45;

    return;
  }
  assert(n >= 1 && n <= max_n);
  srandom((unsigned int)seed * (unsigned int)max_n + (unsigned int)n);
  for (int a = 0; a < n; a++)
    for (int t = 0; t < n; t++)
      cost[a][t] = 3 + (random() % range) + (random() % range) + (random() % range); // [3,3*range]
}

////////////////////////////////////////////////////////////////////////////////////////////////////
//
// code to measure the elapsed time used by a program fragment (an almost copy
// of elapsed_time.h)
//
// use as follows:
//
//   (void)elapsed_time();
//   // put your code to be time measured here
//   dt = elapsed_time();
//   // put morecode to be time measured here
//   dt = elapsed_time();
//
// elapsed_time() measures the CPU time between consecutive calls
//

#if defined(__linux__) || defined(__APPLE__)

//
// GNU/Linux and MacOS code to measure elapsed time
//

#include <time.h>

static double elapsed_time(void) {
  static struct timespec last_time, current_time;

  last_time = current_time;
  if (clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &current_time) != 0)
    return -1.0; // clock_gettime() failed!!!
  return ((double)current_time.tv_sec - (double)last_time.tv_sec) + 1.0e-9 * ((double)current_time.tv_nsec - (double)
      last_time.tv_nsec);
}

#endif

#if defined(_MSC_VER) || defined(_WIN32) || defined(_WIN64)

```

```c
//
// Microsoft Windows code to measure elapsed time
//

#include <windows.h>

static double elapsed_time(void) {
  static LARGE_INTEGER frequency, last_time, current_time;
  static int first_time = 1;

  if (first_time != 0) {
    QueryPerformanceFrequency(&frequency);
    first_time = 0;
  }
  last_time = current_time;
  QueryPerformanceCounter(&current_time);
  return (double)(current_time.QuadPart - last_time.QuadPart) /
         (double)frequency.QuadPart;
}

#endif

////////////////////////////////////////////////////////////////////////////////////////////////////
//
// function to generate a pseudo-random permutation
//

void random_permutation(int n, int t[n]) {
  assert(n >= 1 && n <= 1000000);
  for (int i = 0; i < n; i++)
    t[i] = i;
  for (int i = n - 1; i > 0; i--) {
    int j = (int)floor((double)(i + 1) * (double)random() / (1.0 + (double)RAND_MAX)); // range 0..i
    assert(j >= 0 && j <= i);
    int k = t[i];
    t[i] = t[j];
    t[j] = k;
  }
}

////////////////////////////////////////////////////////////////////////////////////////////////////
//
// place to store best and worst solutions (also code to print them)
//
static int min_cost, min_cost_assignment[max_n]; // smallest cost information
static int max_cost, max_cost_assignment[max_n]; // largest cost information
long histo[60 * max_n + 1];
FILE *f_custos; // CODE FOR EXPORTING HISTOGRAMS AND EXECUTION TIME GRAPHS
FILE *f_reps;   // CODE FOR EXPORTING HISTOGRAMS AND EXECUTION TIME GRAPHS
FILE *n_tested; // CODE FOR EXPORTING HISTOGRAMS AND EXECUTION TIME GRAPHS
FILE *n_time;   // CODE FOR EXPORTING HISTOGRAMS AND EXECUTION TIME GRAPHS
static long n_visited; // number of permutations visited (examined)
// place your histogram global variable here
static double cpu_time;

#define minus_inf -1000000000 // a very small integer
#define plus_inf +1000000000  // a very large integer

static void reset_solutions(void) {
  min_cost = plus_inf;
  max_cost = minus_inf;
  n_visited = 0l;
  memset(histo, 0, sizeof histo);
  cpu_time = 0.0;
}

#define show_info_1 (1 << 0)
#define show_info_2 (1 << 1)
#define show_costs (1 << 2)
#define show_min_solution (1 << 3)
#define show_max_solution (1 << 4)
#define show_histogram (1 << 5)
#define show_all (0xFFFF)

static void show_solutions(int n, char *header, int what_to_show) {

  printf("%s\n", header);
  if ((what_to_show & show_info_1) != 0) {
    printf("  seed .......... %d\n", seed);
    printf("  n ............. %d\n", n);
  }
  if ((what_to_show & show_info_2) != 0) {
    printf("  visited ....... %ld\n", n_visited);
    printf("  cpu time ...... %.3fs\n", cpu_time);
    // n_time = fopen("n_time", "a");
    // n_tested = fopen("n_tested", "a");
    // if (n_time == NULL || n_tested == NULL) {
    //   printf("Error opening file!");
    //   exit(1);
    // } else {
    //   fprintf(n_tested, "%d\n", n);
    //   fprintf(n_time, "%f\n", cpu_time);
    //   fclose(n_time);
    //   fclose(n_tested);
    // }
  }
  if ((what_to_show & show_costs) != 0) {
    printf("  costs ......... (ommited)\n");
    // for (int a = 0; a < n; a++) {
    //   for (int t = 0; t < n; t++)
    //     printf(" %2d", cost[a][t]);
    //   printf("\n%s", (a < n - 1) ? "                 " : "");
    // }
  }
  if ((what_to_show & show_min_solution) != 0) {
    printf("  min cost ...... %d\n", min_cost);
    if (min_cost != plus_inf) {
      printf("  assignement ...");
      for (int i = 0; i < n; i++)
        printf(" %d", min_cost_assignment[i]);
      printf("\n");
    }
  }
  if ((what_to_show & show_max_solution) != 0) {
    printf("  max cost ...... %d\n", max_cost);
    if (max_cost != minus_inf) {
      printf("  assignement ...");
      for (int i = 0; i < n; i++)
        printf(" %d", max_cost_assignment[i]);
      printf("\n");
    }
  }
  if ((what_to_show & show_histogram) != 0) {

    // char str_custos[12];          // WE DECIDED TO EXPORT ALL THE
    // char str_reps[12];            // NEEDED DATA TO SEPARATE FILES AND
    // sprintf(str_custos, "%d", n); // THEN LOAD IT IN OCTAVE. ALL THE
    // sprintf(str_reps, "%d", n);   // FIGURES IN THE REPORT WERE PRODUCED
    // strcat(str_custos, "_costs"); // WITH OCTAVE
    // strcat(str_reps, "_reps");
    // f_custos = fopen(str_custos, "w+");
    // f_reps = fopen(str_reps, "w+");
    // printf("%d %d ", min_cost, max_cost);
    // if (f_custos == NULL || f_reps == NULL) {
    //   printf("Error opening file!");
    //   exit(1);
    // } else {
    //   printf("%d %d", min_cost, max_cost);
    //   for (int i = 42; i <= 840; i++) {
    //     fprintf(f_custos, "%d\n", i);
    //     fprintf(f_reps, "%ld\n", histo[i]);
    //   }
    //   fclose(f_custos);
    //   fclose(f_reps);
    // }
  }
}

////////////////////////////////////////////////////////////////////////////////////////////////////
//
// code used to generate all permutations of n objects
//
// n ........ number of objects
// m ........ index where changes occur (a[0], ..., a[m-1] will not be changed)
```

```c
// a[idx] ... the number of the object placed in position idx
//
// TODO: modify the following function to solve the assignment problem
//

static void generate_all_permutations(int n, int m, int a[n]) {
  if (m < n - 1) {
    for (int i = m; i < n; i++) {
#define swap(i, j)                                                          \
  do {                                                                      \
    int t = a[i];                                                           \
    a[i] = a[j];                                                            \
    a[j] = t;                                                               \
  } while (0)
      swap(i, m);                            // exchange a[i] with a[m]
      generate_all_permutations(n, m + 1, a); // recurse
      swap(i, m);                            // undo the exchange of a[i] with a[m]
#undef swap
    }
  }
  else {
    int temp_cost = 0;
    n_visited++;
    for (int x = 0; x < n; x++) {     // GET THE TOTAL COST OF A SINGLE PERMUTATION
      temp_cost += cost[x][a[x]];     // GET THE TOTAL COST OF A SINGLE PERMUTATION
    }
    // histo[temp_cost] = histo[temp_cost] + 1;  // UPDATE HISTOGRAM
    if (temp_cost < min_cost) {              // CHECK IF THIS PERMUTATION IS THE CHEAPEST SO FAR
      min_cost = temp_cost;                  // IF IT IS, UPDATE THE CHEAPEST PERMUTATION TO THIS ONE
      for (int y = 0; y < n; y++) {          // UPDATE THE ASSIGNMENT
        min_cost_assignment[y] = a[y];
      }
    } else {                                 // IF IT'S NOT THE CHEAPEST, IT MAY BE THE MORE EXPENSIVE ONE
      if (temp_cost > max_cost) {            // CHECK THAT
        max_cost = temp_cost;                // IF IT IS, UPDATE THE MORE EXPENSIVE PERMUTATION TO THIS ONE
        for (int t = 0; t < n; t++) {        // UPDATE THE ASSIGNMENT
          max_cost_assignment[t] = a[t];
        }
      }
    }
  }
}

static void generate_all_permutations_branch_and_bound_min(int n, int m, int a[n], int medium_cost) {
  if (m < n - 1) {
    if (min_cost < medium_cost + 3 * (n - m)) { // CHECK IF, SO FAR, THIS PERMUTATION IS ALREADY
                                                // CONDEMNED// TO BE MORE EXPENSIVE THAN THE CHEAPEST
      return;                                   // SO FAR, AND IF IT IS, DISCARD IT.
    }
    for (int i = m; i < n; i++) {
#define swap(i, j)                                                          \
  do {                                                                      \
    int t = a[i];                                                           \
    a[i] = a[j];                                                            \
    a[j] = t;                                                               \
  } while (0)
      swap(i, m);                      // exchange a[i] with a[m]
      generate_all_permutations_branch_and_bound_min(n, m + 1, a, medium_cost + cost[m][a[m]]); // UPDATE PERMUTATION
       COST SO FAR
      swap(i, m);                      // undo the exchange of a[i] with a[m]
#undef swap
    }
    return;
  }

  else {
    int total_cost = medium_cost + cost[m][a[m]]; // UPDATE PERMUTATION COST
    n_visited++;
    // histo[total_cost] = histo[total_cost] + 1;    // UPDATE HISTOGRAM
    if (total_cost <
        min_cost) {                              // CHECK IF THIS PERMUTATION IS THE CHEAPEST SO FAR
      min_cost = total_cost;
      for (int y = 0; y < n; y++) {              // UPDATE THE ASSIGNMENT
        min_cost_assignment[y] = a[y];
      }
    }
  }
```

```
}

static void generate_all_permutations_branch_and_bound_max(int n, int m, int a[n], int medium_cost) {
  if (m < n - 1) {
    if (max_cost > medium_cost + 60 * (n - m)) { // CHECK IF, SO FAR, THIS PERMUTATION IS ALREADY
                                                 // CONDEMNEDO BE LESS EXPENSIVE THAN THE MOST EXPENSIVE
      return;                                    // SO FAR, AND IF IT IS, DISCARD IT.
    }
    for (int i = m; i < n; i++) {
#define swap(i, j)                                                             \
  do {                                                                         \
    int t = a[i];                                                              \
    a[i] = a[j];                                                               \
    a[j] = t;                                                                  \
  } while (0)
      swap(i, m); // exchange a[i] with a[m]
      generate_all_permutations_branch_and_bound_max(
          n, m + 1, a,
          medium_cost + cost[m][a[m]]); // UPDATE PERMUTATION COST SO FAR
      swap(i, m);                       // undo the exchange of a[i] with a[m]
#undef swap
    }
    return;
  }

  else {
    int total_cost = medium_cost + cost[m][a[m]]; // UPDATE PERMUTATION COST
    // histo[total_cost] = histo[total_cost] + 1;    // UPDATE HISTOGRAM
    n_visited++;
    if (total_cost >  max_cost) {     // CHECK IF THIS PERMUTATION IS THE MOST EXPENSIVE SO FAR
      max_cost = total_cost;
      for (int y = 0; y < n; y++) {   // UPDATE THE ASSIGNMENT
        max_cost_assignment[y] = a[y];
      }
    }
  }
}

static void generate_random_permutations(int n) {
  int arr_perm[n];
  for (int count = 0; count < 1000000; count++) {
    random_permutation(n, arr_perm);  // UMA RANDOM PERMUTATION
    int temp_cost = 0;
    for (int x = 0; x < n; x++) {
      temp_cost += cost[x][arr_perm[x]];  // CUSTO DA P E R M U T A  O
    }
    // histo[n_visited] = min_cost; // UPDATE HISTOGRAM
    n_visited++;
    if (temp_cost < min_cost) {  // VER SE    A P E R M U T A  O COM CUSTO M N I M O
      min_cost = temp_cost;
      for (int y = 0; y < n; y++) {
        min_cost_assignment[y] = arr_perm[y];
      }
    }
    if (temp_cost > max_cost) {
      max_cost = temp_cost;
      for (int t = 0; t < n; t++) {
        max_cost_assignment[t] = arr_perm[t];
      }
    }
  }
}

////////////////////////////////////////////////////////////////////////////////////////////////////
//
// main program
//

int main(int argc, char **argv) {
  if (argc == 2 && argv[1][0] == '-' && argv[1][1] == 'e') {
    seed = 0;
    {
      memset(histo, 0, sizeof(histo));
      int n = 3;
      init_costs(-3); // costs for the example with n = 3
      int a[n];
      for (int i = 0; i < n; i++)
```

```
        a[i] = i;
      reset_solutions();
      (void)elapsed_time();
      generate_all_permutations(n, 0, a);
      cpu_time = elapsed_time();
      show_solutions(n, "Example for n=3", show_all);
      printf("\n");
    }
    {
      memset(histo, 0, sizeof(histo));
      int n = 5;
      init_costs(-5); // costs for the example with n = 5
      int a[n];
      for (int i = 0; i < n; i++)
        a[i] = i;
      reset_solutions();
      (void)elapsed_time();
      generate_all_permutations(n, 0, a);
      cpu_time = elapsed_time();
      show_solutions(n, "Example for n=5", show_all);
      return 0;
    }
  }
  if (argc == 2) {
    seed = atoi(argv[1]); // seed = student number
    if (seed >= 0 && seed <= 1000000) {
      for (int n = 1; n <= max_n; n++) {
        memset(histo, 0, sizeof(histo));
        init_costs(n);
        show_solutions(n, "Problem statement", show_info_1 | show_costs);

        if (n < 14)
        {
          int a[n];
          for(int i = 0;i < n;i++)
          a[i] = i; // initial permutation
          reset_solutions();
          (void)elapsed_time();
          generate_all_permutations(n,0,a);      // DONE WITH BRUTE FORCE SOLUTION
          cpu_time = elapsed_time();
          show_solutions(n,"Brute force", show_info_2 | show_min_solution | show_max_solution);
          reset_solutions();
          (void)elapsed_time();
          generate_random_permutations(n);   // DONE WITH RANDOM PERMUTATIONS
          cpu_time = elapsed_time();
          show_solutions(n,"Random Permutations", show_info_2 | show_min_solution | show_max_solution);
        }

        if (n < 16)
        {
          int a[n];
          for (int i = 0; i < n; i++)
            a[i] = i; // initial permutation
          reset_solutions();
          (void)elapsed_time();
          generate_all_permutations_branch_and_bound_max(n, 0, a, 0); // DONE WITH BRUTE FORCE BRANCH-AND-BOUND SOLUTION
          cpu_time = elapsed_time();
          show_solutions(n, "Brute force with branch-and-bound max",show_max_solution | show_info_2);
          reset_solutions();
          (void)elapsed_time();
          generate_all_permutations_branch_and_bound_min(n, 0, a, 0); // DONE WITH BRUTE FORCE BRANCH-AND-BOUND SOLUTION
          cpu_time = elapsed_time();
          show_solutions(n, "Brute force with branch-and-bound min",show_min_solution | show_info_2);
        }
        printf("\n");
      }
      return 0;
    }
  }
  fprintf(stderr, "usage: %s -e              # for the examples\n", argv[0]);
  fprintf(stderr, "usage: %s student_number\n", argv[0]);
  return 1;
}
```