Projeto 2
Random ordered trees

Diogo Andrade, NMec 89265, contributo:
Pedro Escaleira, NMec 88821, contributo:
Rafael Simões, NMec 88984, contributo:

1. Índice

Índice **2**

Introdução **3**

Resultados obtidos **4**

Tabela de dados para 1000000 árvores aleatórias 4

Altura máxima da árvore binária 6

Comentários acerca dos resultados obtidos 6

Número de folhas da árvore binária 8

Comentários acerca dos resultados obtidos 8

Custo médio de procura de um item de informação contido na árvore binária 10

Comentários acerca dos resultados obtidos 10

Custo médio de procura de um item de informação não contido na árvore binária 12

Comentários acerca dos resultados obtidos 12

Problema final 14

Conclusão 15

Código **18**

binary_tree.c 18

graficos.m 26

compare_heights.m 28

compare_hits.m 29

compare_miss.m 30

stg_g.m 31

2. Introdução

O propósito do segundo trabalho prático é estudar árvores binárias ordenadas aleatoriamente e a forma como “se comportam” quando são inseridos elementos nela desta forma. Em particular, para uma árvore com n nós, calcular a altura média máxima da árvore, o número médio de folhas, o custo médio de procura de um item presente na árvore e o custo médio de procura de um item não presente na árvore.

Sendo assim, este estudo é baseado numa simulação, pelo que todos os resultados obtidos não devem ser tomados como absolutos, ou seja, os resultados podem variar de outra forma quando executado novamente o programa. Contudo, por norma, estes tendem a permanecer praticamente constantes ao longo das várias execuções, dada a elevada quantidade de simulações que fazemos para cada número de nós das árvores binárias.

3. Resultados obtidos

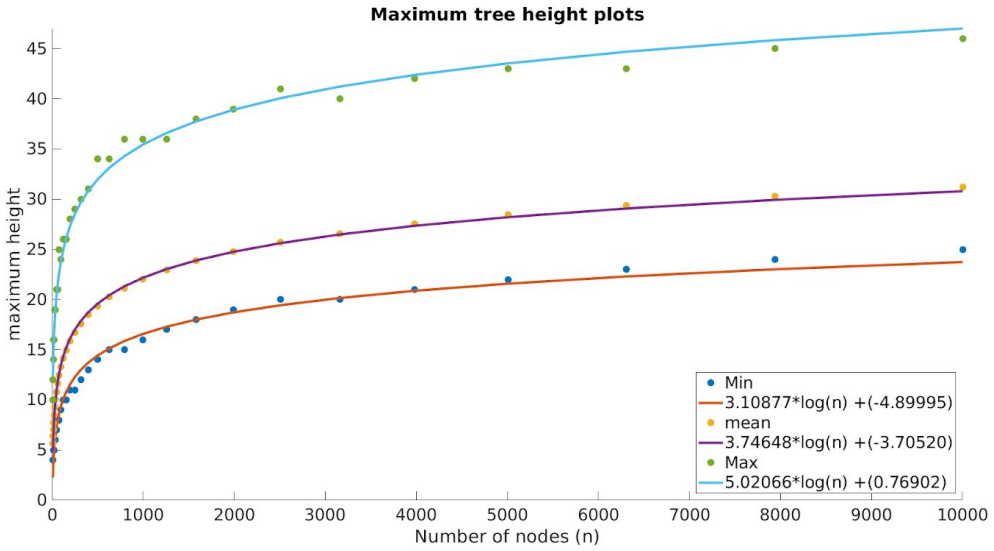
3.1. Tabela de dados para 1000000 árvores aleatórias

	maximum tree height				number of leaves				calls on hit		calls on miss	
n	min	max	mean	std	min	max	mean	std	mean	std	mean	std
10	4	10	5.6396	0.9147	1	5	3.6680	0.6995	3.4434	0.3932	5.0395	0.3574
13	4	12	6.4057	1.0140	2	7	4.6671	0.7889	3.8495	0.4312	5.5031	0.4004
16	5	14	7.0386	1.0882	2	8	5.6671	0.8688	4.1840	0.4571	5.8791	0.4302
20	5	16	7.7404	1.1642	3	10	6.9989	0.9660	4.5554	0.4828	6.2908	0.4598
25	5	16	8.4613	1.2350	4	13	8.6678	1.0743	4.9371	0.5048	6.7087	0.4854
32	6	19	9.2804	1.3101	6	16	10.9970	1.2121	5.3705	0.5271	7.1775	0.5111
40	6	19	10.0377	1.3700	7	19	13.6692	1.3507	5.7703	0.5429	7.6051	0.5297
50	7	21	10.8046	1.4275	10	24	16.9979	1.5059	6.1743	0.5566	8.0336	0.5457
63	8	21	11.6190	1.4851	13	29	21.3340	1.6880	6.6027	0.5693	8.4839	0.5604
79	8	25	12.4287	1.5345	18	35	26.6638	1.8861	7.0278	0.5806	8.9275	0.5734
100	9	24	13.2811	1.5852	23	44	33.6620	2.1186	7.4761	0.5924	9.3922	0.5866
126	10	26	14.1306	1.6306	31	53	42.3303	2.3775	7.9184	0.6003	9.8481	0.5956
158	10	26	14.9698	1.6712	40	64	52.9942	2.6594	8.3565	0.6084	10.2977	0.6046
200	11	28	15.8505	1.7121	52	80	67.0028	2.9889	8.8143	0.6154	10.7655	0.6124
251	11	29	16.7081	1.7494	68	100	83.9987	3.3507	9.2570	0.6211	11.2163	0.6186
316	12	30	17.5834	1.7803	87	123	105.6610	3.7561	9.7090	0.6267	11.6753	0.6247
398	13	31	18.4641	1.8109	113	152	132.9943	4.2104	10.1620	0.6288	12.1341	0.6272
501	14	34	19.3483	1.8398	146	189	167.3329	4.7240	10.6161	0.6327	12.5930	0.6314
631	15	34	20.2415	1.8669	186	238	210.6705	5.2979	11.0721	0.6352	13.0530	0.6341
794	15	36	21.1398	1.8906	237	293	264.9925	5.9409	11.5285	0.6378	13.5127	0.6370
1000	16	36	22.0410	1.9165	302	364	333.6633	6.6689	11.9858	0.6400	13.9728	0.6394

1259	17	36	22.9464	1.9351	382	458	419.9879	7.4811	12.4433	0.6411	14.4326	0.6405
1585	18	38	23.8498	1.9567	486	570	528.6655	8.4022	12.9012	0.6420	14.8924	0.6416
1995	19	39	24.7649	1.9747	616	715	665.3311	9.4187	13.3595	0.6431	15.3523	0.6428
2512	20	41	25.6779	1.9898	787	888	837.6626	10.5744	13.8198	0.6445	15.8139	0.6443
3162	20	40	26.5921	2.0087	997	1111	1054.3432	11.8558	14.2777	0.6444	16.2729	0.6442
3981	21	42	27.5134	2.0254	1262	1395	1327.3409	13.2946	14.7380	0.6461	16.7340	0.6460
5012	22	43	28.4388	2.0399	1598	1750	1671.0024	14.9376	15.1969	0.6458	17.1937	0.6457
6310	23	43	29.3603	2.0565	2020	2179	2103.7030	16.7642	15.6577	0.6471	17.6551	0.6470
7943	24	45	30.2889	2.0678	2562	2729	2647.9894	18.7925	16.1180	0.6476	18.1158	0.6475
10000	25	46	31.2185	2.0799	3235	3431	3333.6712	21.1021	16.5781	0.6481	18.5764	0.6480

3.2. Altura máxima da árvore binária

3.2.1. Comentários acerca dos resultados obtidos



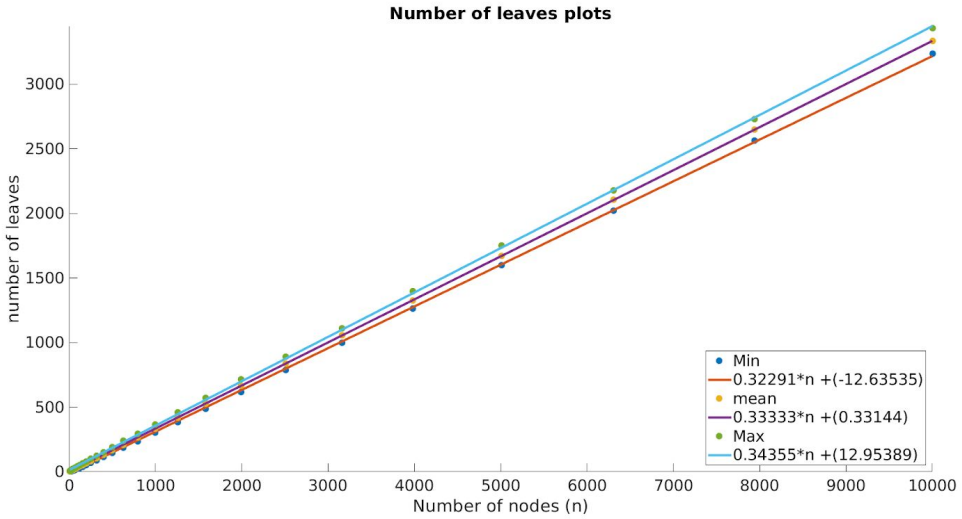
Como podemos observar pelo gráfico obtido, os tamanhos mínimos e médios da árvore binária seguiram um crescimento logarítmico, algo que é facilmente compreensível numa árvore binária balanceada dada a forma como os dados estão armazenados nesta: cada nó pode ter até dois “nós filhos”, pelo que cada nível da árvore, se completamente preenchido, apresenta o dobro dos elementos do nível anterior.

Contudo, os tamanhos máximos obtidos seguem também um crescimento logarítmico, algo que não estávamos à espera, já que, no caso de se inserirem por ordem os elementos na árvore (árvore com tamanho máximo possível), esta vai possuir uma altura igual ao número de elementos inseridos, pelo que o crescimento deste deveria ser linear. Depois de refletirmos sobres este resultado, chegámos à conclusão de que é normal o termos obtido, já que estas situações ocorrem em muito raras situações e nós apenas geramos um número relativamente pequeno de árvores.

Para além disso, podemos detectar que a curva dos valores médios de tamanhos das árvores se encontra mais próxima da curva dos tamanhos mínimos do que da dos tamanhos máximos, o que nos leva a crer que as árvores binárias aleatórias tendem a ter comprimentos menores do que maiores.

3.3. Número de folhas da árvore binária

3.3.1. Comentários acerca dos resultados obtidos

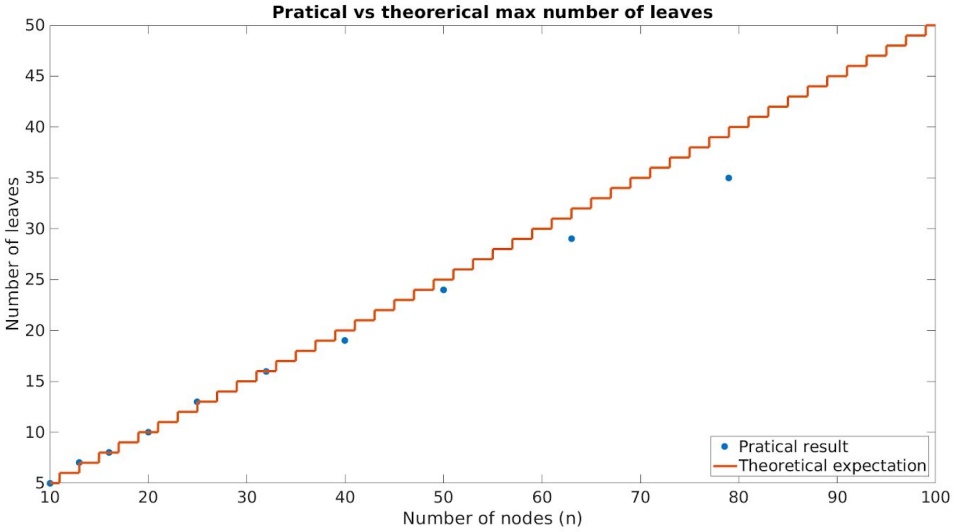


Dos gráficos acima obtidos após a execução do programa, percebemos que o número de folhas “cresce” de acordo com o número de nós da árvore binária de forma linear em árvores binárias aleatórias.

Poderíamos esperar que o número mínimo de folhas da árvore binária seria sempre 1, pelo facto da inserção ordenada dos elementos na árvore binária ser o caso em que temos um número mínimo de folhas, mas como já vimos nos resultados do ponto anterior, essa situação é extremamente rara, principalmente quando o número de nós da árvore é mais elevado.

Para os valores máximos do número de folhas, também os resultados obtidos se afastaram daquilo que teoricamente pensamos que iria acontecer. De acordo com os nossos cálculos, o número de folhas máximos que se pode obter para um determinado número de nós obtém-se quando a árvore é balanceada, sendo que nesta situação o número de folhas é dado pela

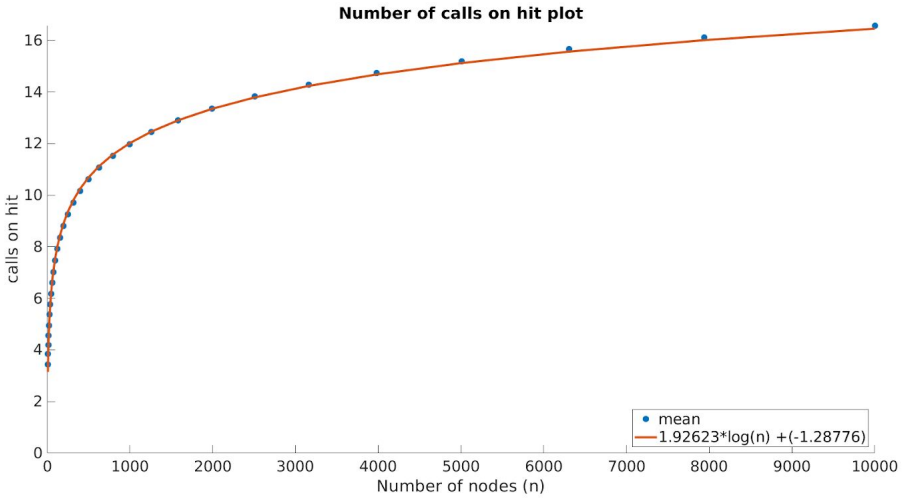
fórmula $\left\lfloor \frac{n+1}{2} \right\rfloor$. Mas como podemos ver no seguinte gráfico, apenas para os menores números de nós da árvore é que verificamos esta equação (excluímos deste gráfico os resultados obtidos para um maior número de nós para uma mais facilitada comparação):



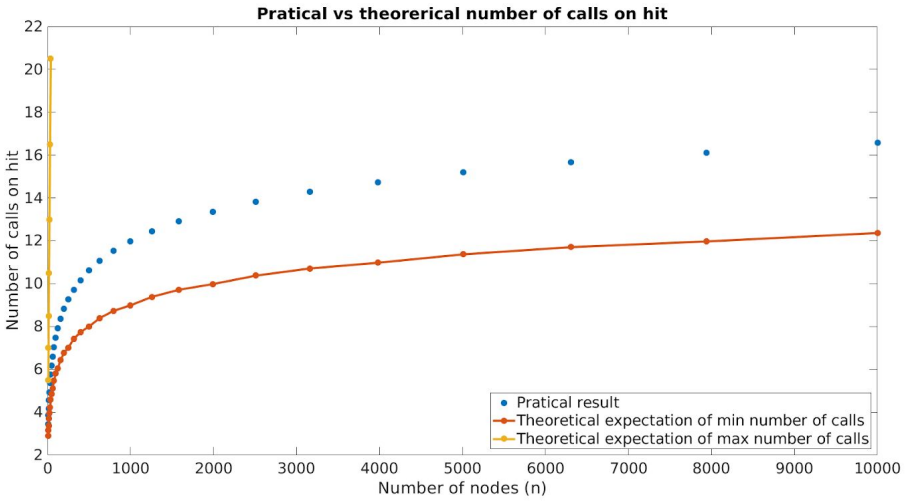
Também estes resultados podem ser explicados pela extremamente baixa probabilidade de se formarem árvores binárias balanceadas, probabilidade esta que é menor para um número de nós mais elevados.

3.4. Custo médio de procura de um item de informação contido na árvore binária

3.4.1. Comentários acerca dos resultados obtidos



Como podemos notar, o número médio de chamadas com êxito apresenta um comportamento logarítmico em relação ao número de nós da árvore binária aleatória, comportamento que esperaríamos numa árvore balanceada. Como podemos perceber no seguinte gráfico, há de facto uma grande proximidade entre os valores obtidos na prática e os valores que se esperam obter para árvores balanceadas (que são as que possuem o mínimo de chamadas com êxito possível para qualquer número de nós):

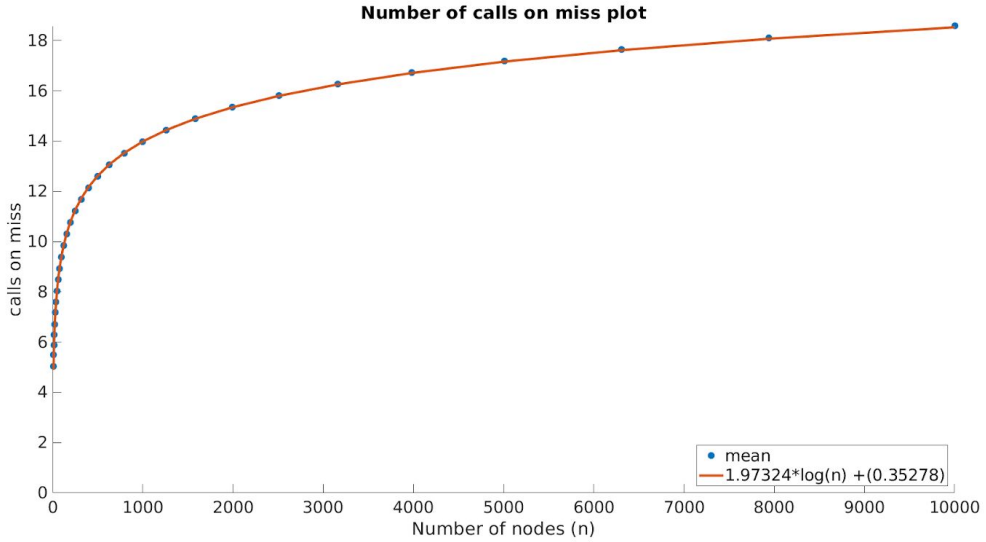


Nesta figura foi-nos impossível representar todos os valores do número máximo possível de chamadas com sucesso (quando todos os elementos presentes na árvore são inseridos por ordem) que se esperam obter para cada número de nós, já que este apresenta um crescimento linear em relação ao número de nós da árvore, pelo que apresentamos apenas os primeiros 7 valores.

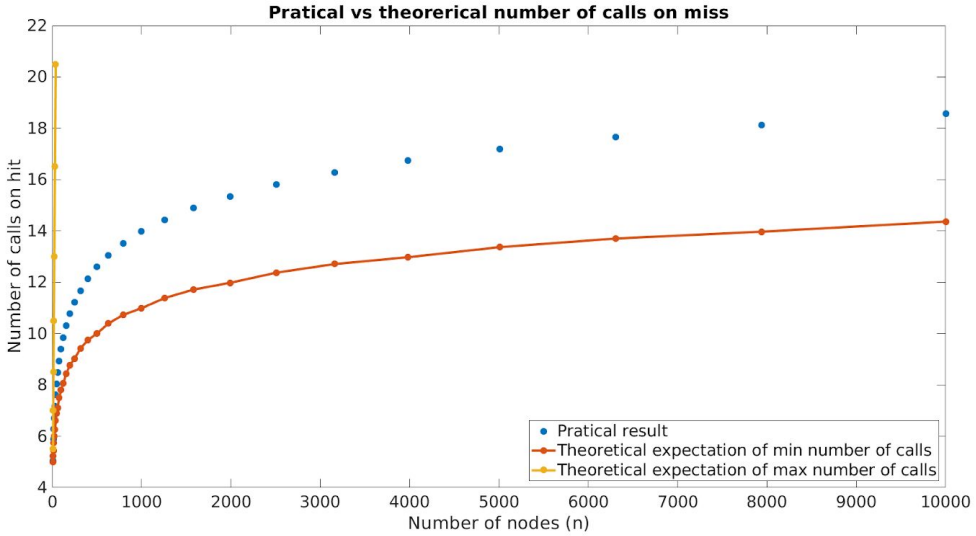
Sendo assim, podemos concluir que o número médio de chamadas com sucesso em árvores binárias aleatórias tende a ter valores menores, sendo estes muito próximos dos valores obtidos para árvores balanceadas e que uma chamada com êxito numa árvore deste tipo tem uma complexidade computacional logarítmica.

3.5. Custo médio de procura de um item de informação não contido na árvore binária

3.5.1. Comentários acerca dos resultados obtidos



Pelos resultados obtidos, podemos compreender que o número médio de chamadas sem êxito apresenta um crescimento logarítmico em relação ao número de nós em árvores binárias aleatórias, comportamento este característico duma árvore binária balanceada. Da mesma forma que para o número de chamadas com êxito, os valores obtidos encontram-se muito próximos dos mínimos valores possíveis do número de chamadas com êxito que uma árvore binária pode apresentar (árvores binárias balanceadas):



Tal como no ponto anterior, reduzimos o número máximo teórico de chamadas sem êxito dada a impossibilidade de visualização dos restantes gráficos caso não o fizéssemos.

Desta maneira, concluímos também que o número médio de chamadas sem êxito tende a apresentar valores menores e próximos dos valores das árvores binárias balanceadas e que uma chamada sem êxito tem uma complexidade computacional logarítmica.

3.6. Problema final

Foi nos proposto explicar como modificar o custo médio computacional de *count on miss* quando se inserem os números 1^2, 5^2, 5^2, ..., $(2n-1)^2$, ao invés de 1, 3, 5, ..., (2n-1), assumindo que estamos a procurar por números 1, 2, 3, ..., 2n, (2n+1). Ao fim de analisarmos o código e a topologia do problema, verificamos que uma solução seria verificar, na função ***search*** *,* se a raiz quadrada do valor a ser procurado é um valor inteiro ímpar; caso seja, a função prossegue com as seguintes instruções; caso contrário, passamos para o próximo nó da árvore binária.

Desta forma, conseguimos reduzir a complexidade computacional em termos temporais, já que limitamos os valores de pesquisa aos que correspondem à estrutura da árvore binária (que contém apenas os quadrados de número ímpares).

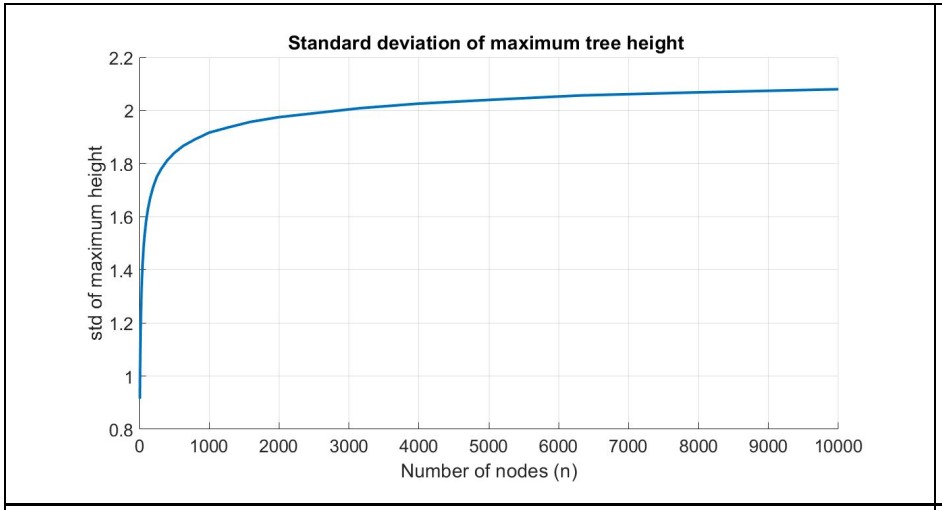
3.7. Conclusão

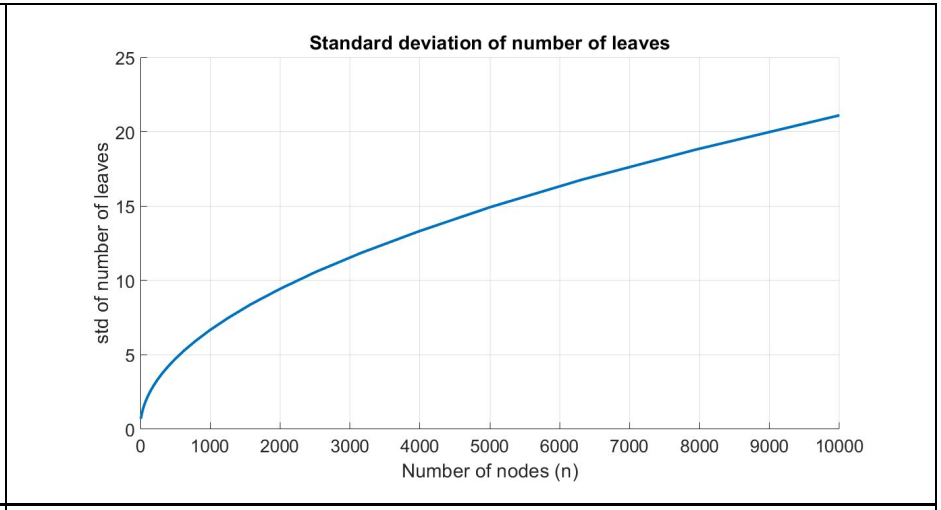
Como podemos observar ao longo do relatório, as árvores binárias aleatórias formadas tendem a ter uma estrutura mais próxima de uma árvore binária balanceada (em detrimento de árvores binárias onde os elementos são inseridos por ordem). É relativamente compreensível termos obtido estes resultados se calcularmos a probabilidade de se formarem árvores balanceadas e árvores onde os elementos foram inseridos por ordem:

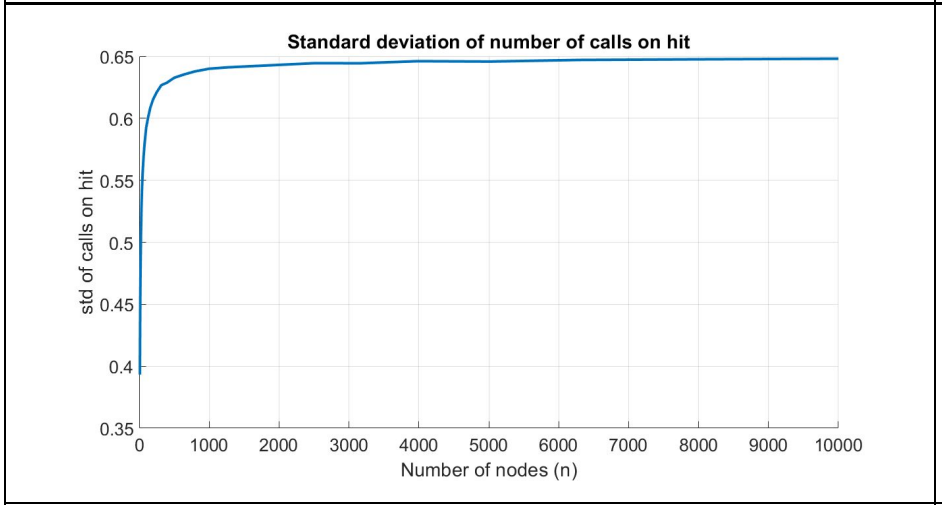
$\frac{2}{n!}$ Probabilidade de se obter uma árvore onde os elementos foram inseridos por ordem	$\frac{1! * 2! * 4! * ... * (2^{\lfloor log_2(n+1) \rfloor - 1}))! * A_{2^{\lceil log_2(n+1) \rceil - 1}, (n-1-2-4-...-(2^{\lfloor log_2(n+1) \rfloor - 1}))}}{n!}$ Probabilidade de se obter uma árvore balanceada

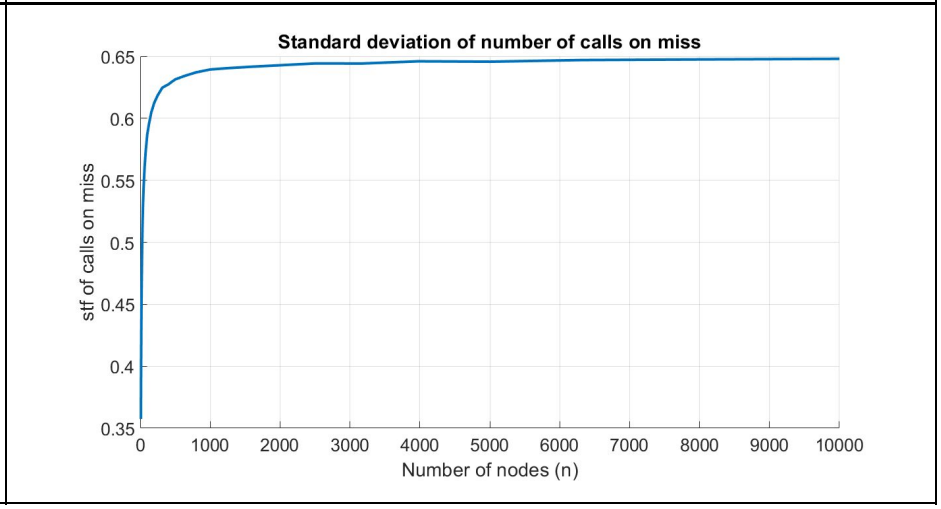
Onde a segunda probabilidade é claramente muito maior que a segunda para valores de **n** elevados. Através destas duas fórmulas podemos verificar, por exemplo para 10 nós, que a probabilidade de se obterem árvores onde os elementos foram inseridos por ordem é igual a aproximadamente 0.000000551 (0.55 árvores em 1000000) e para o segundo caso é aproximadamente 0.0044445 (4444.5 árvores em 1000000), mas para 13 nós, estas já são, respectivamente, 0.000000000132 (0.0003 árvores em 1000000) e 0,0001554 (155.4 árvores em 1000000). Ou seja, como podemos perceber é mais provável formarem-se árvores “quase ou mesmo balanceadas” e muito pouco provável formarem-se árvores onde os elementos foram inseridos “quase ou mesmo todos por ordem”.

Para além destes resultados, ainda obtivemos os valores dos desvios padrões para cada uma das quatro experiências, onde percebemos que os resultados de cada uma das experiências se encontram relativamente próximos da média de cada uma, pelo facto destes valores serem relativamente pequenos:


Standard deviation of maximum tree height
std of maximum height
Number of nodes (n)
2.2
2
1.8
1.6
1.4
1.2
1
0.8
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000


Standard deviation of number of leaves
std of number of leaves
Number of nodes (n)
25
20
15
10
5
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000


Standard deviation of number of calls on hit
std of calls on hit
Number of nodes (n)
0.65
0.6
0.55
0.5
0.45
0.4
0.35
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000


Standard deviation of number of calls on miss
stf of calls on miss
Number of nodes (n)
0.65
0.6
0.55
0.5
0.45
0.4
0.35
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000

Concluindo, com o estudo feito neste trabalho prático, entendemos como é que árvores binárias aleatórias tendem a ser formadas e que estruturas é que aproximadamente apresentam: geometricamente. tendem a ser mais “dispersas” e menos lineares, com alturas e número de chamadas com êxito e sem êxito próximas dos mínimos possíveis e com um número de folhas próximo dos máximos possíveis.

4. Código

4.1. binary_tree.c

Este código foi nos disponibilizado numa das aulas práticas da disciplina e foi a base deste projeto, sendo-nos apenas necessário completar quatro funções e acrescentar a possibilidade de armazenar a informação necessária em ficheiros.
Este código suporta funções de procura de informação e cálculo quer da altura de uma árvore binária, quer do número de folhas desta e do custo médio de procura por informação contida e não contida na nesta.

```
//
// Tomás Oliveira e Silva, AED,  November 2018
//
// TO DO: Place your own identification here
//
// empirical study of random ordered binary trees
//

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

//
// each node of our ordered binary tree will store a long integer
//
// the root of the tree should be declared as follows (set initially to an empty tree):
//
//   tree_node *root = NULL;
//

typedef struct tree_node
{
  struct tree_node *left;   // pointer to the left branch (a sub-tree)
  struct tree_node *right;  // pointer to the right branch (a sub-tree)
  struct tree_node *parent; // pointer to the parent node (NULL for the root of the tree)
  long data;                    // the data item (we use a long here)
} tree_node;

//
// insert a node in the tree (it is assumed that the tree does not store repeated data)
//
// use it as follows (example):
//
//   insert_node(&root,&new_node);
//
```

```c
static void insert_node(tree_node **link, tree_node *n)
{
  tree_node *parent;

  parent = NULL;
  while (*link != NULL)
  {
          if (n->data == (*link)->data)
          {
          fprintf(stderr, "insert_node: %ld is already in the tree\n", n->data);
          exit(1);
          }
          parent = *link;
          link = (n->data < (*link)->data) ? &((*link)->left) : &((*link)->right); // select branch
  }
  *link = n;
  n->parent = parent;
  n->left = n->right = NULL;
}

//
// count the number of leaves of the tree
//
// use if as follows (example):
//
//   int n_leaves = count_leaves(root);
//

static int count_leaves(tree_node *link)
{
  int num_leaves = 0;
  if (link->left != NULL || link->right != NULL)
  {
          if (link->left != NULL)
          num_leaves += count_leaves(link->left);
          if (link->right != NULL)
          num_leaves += count_leaves(link->right);
  }
  else
          num_leaves++;
  return num_leaves;
}

//
// compute the height of the tree
//
// use if as follows (example):
//
//   int height = tree_height(root);
//

static int tree_height(tree_node *link)
```

```
{
  int actual_height = 1, height = 1;

  if (link->right != NULL)
          actual_height += tree_height(link->right);
  if (actual_height > height)
          height = actual_height;
  actual_height = 1;
  if (link->left != NULL)
          actual_height += tree_height(link->left);
  if (actual_height > height)
          height = actual_height;

  return height;
}

//
// recursive function used to search for the location of a data item
//
// use if as follows (example):
//
//   tree_node *node = search_tree(root,data);
//

static int search_counter;

tree_node *search_tree(tree_node *link, long data)
{
  search_counter++;
  if (link == NULL)
          return NULL;
  if (link->data == data)
          return link;
  return search_tree((data < link->data) ? link->left : link->right, data);
}

//
// assuming that each data item is searched for with equal probability, compute the average number
// of recursive function calls to the search_tree() function when
// 1) the search is successful (a hit)
// 2) the search is not successful (a miss)
//
// use them as follows (example):
//
//   double average_calls_on_hit = (double)count_function_calls_on_hit(root,0) / (double)number_of_nodes;
//   double average_calls_on_miss = (double)count_function_calls_on_miss(root,0) / (double)number_of_nodes;
//

static int count_function_calls_on_hit(tree_node *link, int level)
{
  if (link == NULL)
          return 0;
  return level + 1 +
```

```
          count_function_calls_on_hit(link->left, level + 1) +
          count_function_calls_on_hit(link->right, level + 1);
}

static int count_function_calls_on_miss(tree_node *link, int level)
{
  if (link == NULL)
          return level + 1;
  return count_function_calls_on_miss(link->left, level + 1) +
          count_function_calls_on_miss(link->right, level + 1);
}

//
// random permutation of the n numbers 1, 3, 5, ..., 2*n-1
//
// use if as follows (example):
//
//   int n = 100;
//   int a[n];
//   rand_perm(n,&a[0]);
//

static void rand_perm(int n, int *a)
{
  int i, j, k;

  for (i = 0; i < n; i++)
          a[i] = 2 * i + 1;
  for (i = n - 1; i > 0; i--)
  {
          j = (int)floor((double)(i + 1) * (double)rand() / (1.0 + (double)RAND_MAX)); // range 0..i
          k = a[i];
          a[i] = a[j];
          a[j] = k;
  }
}

//
// main program
//

int main(int argc, char **argv)
{
  int details = (argc == 3 && argv[1][0] == '-' && argv[1][1] == 'a' && atoi(argv[2]) > 0) ? 1 : 0;
  int n_experiments = 1000000; // TO DO: use more (1000000 should take 2 to 3 hours)

  srandom(1u); // ensure reproducible results
  printf("                                data for %d random trees\n", n_experiments);
  printf("           maximum tree height             number of leaves               calls on hit          calls on miss\n");
  printf("           ---------------------  ------------------------------  --------------  --------------\n");
  printf("           n  min max mean       std        min   max            mean       std        mean       std        mean       std\n");
  printf("------  --- --- ------- ------  ----- ----- ---------- --------  ------- ------  ------- ------\n");
  FILE *height_data = fopen("height_data.txt", "w");
```

```
FILE *leaves_data = fopen("leaves_data.txt", "w");
FILE *count_hit_data = fopen("count_hit_data.txt", "w");
FILE *count_miss_data = fopen("count_miss_data.txt", "w");
for (int n_log = 1 * 10; n_log <= 4 * 10; n_log++)
{
        int n = (int)round(pow(10.0, (double)n_log / 10.0)); // the number of nodes of the tree
        int a[n];                                            // the nodes' data
        tree_node *root, nodes[n];                           // the root and the storage space for the nodes of the tree
        int h_height[n + 1];                                 // for an histogram of the heights of the random trees
        int h_leaves[n + 1];                                 // for an histogram of the number of leaves of the random trees
        double mean, std;                                    // for mean and standard deviation computations
        double x, hit[2], miss[2];                           // for the average number of hits and misses
        int m, M;                                            // location of minima and maxima

        printf("%6d", n);
        //
        // the example in the slides
        //
        if (n == 10)
        {
        root = NULL;
        nodes[0].data = 3l;
        insert_node(&root, &nodes[0]);
        nodes[1].data = 1l;
        insert_node(&root, &nodes[1]);
        nodes[2].data = 9l;
        insert_node(&root, &nodes[2]);
        nodes[3].data = 7l;
        insert_node(&root, &nodes[3]);
        nodes[4].data = 5l;
        insert_node(&root, &nodes[4]);
        if (count_leaves(root) != 2)
        {
        fprintf(stderr, "count_leaves() returned a wrong value\n");
        exit(1);
        }
        if (tree_height(root) != 4)
        {
        fprintf(stderr, "tree_height() returned a wrong value\n");
        exit(1);
        }
        search_counter = 0;
        for (int i = 1; i <= 9; i += 2)
        if (search_tree(root, (long)i) == NULL)
        return 1; // impossible if the program is correct
        if (count_function_calls_on_hit(root, 0) != search_counter)
        {
        fprintf(stderr, "count_function_calls_of_hit() returned a wrong value\n");
        exit(1);
        }
        search_counter = 0;
        for (int i = 0; i <= 10; i += 2)
        if (search_tree(root, (long)i) != NULL)
```

```
return 1; // impossible if the program is correct
if (count_function_calls_on_miss(root, 0) != search_counter)
{
fprintf(stderr, "count_function_calls_of_miss() returned a wrong value\n");
exit(1);
}
}
//
// the experiments
//
for (int i = 0; i <= n; i++)
h_height[i] = h_leaves[i] = 0;
hit[0] = hit[1] = miss[0] = miss[1] = 0.0;
for (int n_experiment = 0; n_experiment < n_experiments; n_experiment++)
{
rand_perm(n, &a[0]);
root = NULL;
for (int i = 0; i < n; i++)
{
nodes[i].data = (long)a[i];
insert_node(&root, &nodes[i]);
}
h_height[tree_height(root)]++;
h_leaves[count_leaves(root)]++;
x = (double)count_function_calls_on_hit(root, 0) / (double)n; // there are n nodes
hit[0] += x;
hit[1] += x * x;
x = (double)count_function_calls_on_miss(root, 0) / (double)(n + 1); // there are n+1 NULL pointers
miss[0] += x;
miss[1] += x * x;
}

mean = std = 0.0;
m = n + 1;
M = -1;
for (int i = 0; i <= n; i++)
if (h_height[i] != 0)

{
mean += (double)i * (double)h_height[i];
std += (double)i * (double)i * (double)h_height[i];
if (i < m)
m = i;
if (i > M)
M = i;
}
mean /= (double)n_experiments;
std /= (double)n_experiments;
std = sqrt(std - mean * mean);
printf("  %3d %3d %7.4f %6.4f", m, M, mean, std);
fprintf(height_data, "%d %d %d %7.4f %6.4f\n", n, m, M, mean, std);

mean = std = 0.0;
```

```
        m = n + 1;
        M = -1;
        for (int i = 0; i <= n; i++)
        if (h_leaves[i] != 0)
        {
        mean += (double)i * (double)h_leaves[i];
        std += (double)i * (double)i * (double)h_leaves[i];
        if (i < m)
        m = i;
        if (i > M)
        M = i;
        }
        mean /= (double)n_experiments;
        std /= (double)n_experiments;
        std = sqrt(std - mean * mean);
        printf("  %5d %5d %10.4f %8.4f", m, M, mean, std);
        fprintf(leaves_data, "%d %d %d %7.4f %6.4f\n", n, m, M, mean, std);

        mean = hit[0] / (double)n_experiments;
        std = hit[1] / (double)n_experiments;
        std = sqrt(std - mean * mean);
        printf("  %7.4f %6.4f", mean, std);
        fprintf(count_hit_data, "%d %7.4f %6.4f\n", n, mean, std);
        mean = miss[0] / (double)n_experiments;
        std = miss[1] / (double)n_experiments;
        std = sqrt(std - mean * mean);
        printf("  %7.4f %6.4f", mean, std);
        fprintf(count_miss_data, "%d %7.4f %6.4f\n", n, mean, std);

        printf("\n");
        //
        // output the tree height data
        //
        if (details != 0 || n == atoi(argv[2]))
        {
        printf("   i frac height\n");
        printf("  ----- -----------\n");
        for (int i = 0; i <= n; i++)
        if (h_height[i] != 0)
        printf("  %5d %11.9f\n", i, (double)h_height[i] / (double)n_experiments);
        printf("  ----- -----------\n");
        }
        //
        // output the number of leaves data
        //
        if (details != 0 || n == atoi(argv[2]))
        {
        printf("   i frac leaves\n");
        printf("  ----- -----------\n");
        for (int i = 0; i <= n; i++)
        if (h_leaves[i] != 0)
        printf("  %5d %11.9f\n", i, (double)h_leaves[i] / (double)n_experiments);
        printf("  ----- -----------\n");
```

```
            }
            //
            // done
            //
            fflush(stdout);
    }
    printf("------  --- --- ------- ------  ----- ----- ---------- --------  ------- ------  ------- ------\n");
    printf("done in %.1f seconds\n", elapsed_time());
    fclose(height_data);
    fclose(leaves_data);
    fclose(count_hit_data);
    fclose(count_miss_data);
    return 0;
}
```

4.2. graficos.m

Este código matlab é responsável para ilustrar os gráficos dos respetivos “fit” pedidos e também pela conversão desses gráficos em imagem “.png” para poderem ser usadas neste relatório.

```
clear all;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Este script gera imagens png dos graficos
%% o nome dos ficheiros tem que ser iguais aos nomes do array output
display_dots=1; %change it, 1->display dots;0->not display dots
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

format_s="%.5f*Log(n) +(%.5f)";
output=["count_hit_data","count_miss_data","height_data","leaves_data"];
titles=["Number of calls on hit plot","Number of calls on miss plot","Maximum tree height plots","Number of leaves plots"];
ylabels=["calls on hit", "calls on miss", "maximum height", "number of leaves"];
height_and_leaves="%d %d %d %f %f\n";
counts_hit_miss="%d %f %f\n";
format_type=counts_hit_miss;
divisor=3;
for i=1:length(output)
           file_name="";
           file_name=strcat(output(i),'.txt');
           file=fopen(file_name,"r");
           if (i>2)
           format_type=height_and_leaves;
           divisor=5;
           end
           A=fscanf(file,format_type);
           N=length(A);
           average=zeros(N/divisor,1);
           std=zeros(N/divisor,1);
           n_values=zeros(N/divisor,1);
           if(i>2)
           max=zeros(N/divisor,1);
           min=zeros(N/divisor,1);
           end
           k=1;
           for j=1:divisor:N
           n_values(k)=A(j);
           average(k)=A(j+1+rem(divisor,3));
           std(k)=A(j+2+rem(divisor,3));
           if(i>2)
           min(k)=A(j+1);
           max(k)=A(j+2);
           end
           k=k+1;
```

```
        end
        figure('Position',[0 0 1920 1080]); %change this to set resolution of graph
        hold on;
        if(i==4)
        format_s="%.5f*n +(%.5f)";
        D=[n_values,1+0*n_values];
        else
        D=[log(n_values),1+0*n_values];
        end
        w=pinv(D)*average;

        if(i>2)
        if(display_dots==1)
        plot(n_values,min,".", 'LineWidth', 3, 'MarkerSize', 30);
        end
        w_min=pinv(D)*min;
        plot(n_values,D*w_min, 'LineWidth', 3, 'MarkerSize', 30);
        end
        if(display_dots==1)
        plot(n_values,average,".", 'LineWidth', 3, 'MarkerSize', 30);
        end
        plot(n_values,D*w, 'LineWidth', 3, 'MarkerSize', 30);
        if(i>2)
        w_max=pinv(D)*max;
        if(display_dots==1)
        plot(n_values,max,".", 'LineWidth', 3, 'MarkerSize', 30);
        end
        plot(n_values,D*w_max, 'LineWidth', 3, 'MarkerSize', 30);
        if (display_dots==1)
        legend({"Min",sprintf(format_s,w_min(1),w_min(2)),"mean",sprintf(format_s,w(1),w(2)),"Max",sprintf(format_s,w_max(1),w_max(2))}, 'FontSize', 20, 'Location', 'southeast');
        else
        legend({sprintf(format_s,w_min(1),w_min(2)),sprintf(format_s,w(1),w(2)),sprintf(format_s,w_max(1),w_max(2))}, 'FontSize', 20, 'Location', 'southeast');
        end
        else
        if(display_dots==1)
        legend({"mean",sprintf(format_s,w(1),w(2))}, 'FontSize', 20, 'Location', 'southeast');
        else
        legend(sprintf(format_s,w(1),w(2)), 'FontSize', 20, 'Location', 'southeast');
        end
        end
        ylim([0 inf]);
        title(titles(i), 'FontSize', 22);
        xlabel("Number of nodes (n)", 'FontSize', 20);
        ylabel(ylabels(i), 'FontSize', 20);
        set(gca, 'FontSize', 20);

        set(gcf,'PaperPositionMode','auto')
        print(output(i),'-dpng','-r0')
        fclose(file);
end
```

4.3. compare_heights.m

Este código é responsável de apresentar um gráfico para fazer comparação entre os gráficos dos valores das altura altura da árvore obtidos na prática com os valores esperados teoricamente.

```
function compare_heights(file_name, last_n_index)
   file = fopen(file_name, "r");
   data = fscanf(file, "%d %d %d %f %f\n");

   plot(data(1:5:last_n_index*5), data(3:5:last_n_index*5), '.', 'MarkerSize', 30);
   hold on;
   plot([data(1):0.01:data(last_n_index*5 + 1)], floor(([data(1):0.01:data(last_n_index*5 + 1)] + 1)/2), 'LineWidth', 3);
   title("Pratical vs theorerical max number of leaves", 'FontSize', 22);
   xlabel("Number of nodes (n)", 'FontSize', 20);
   ylabel("Number of leaves", 'FontSize', 20);
   legend({"Pratical result", "Theoretical expectation"}, 'FontSize', 20, 'Location', 'southeast');
   set(gca, 'FontSize', 20);
end
```

4.4. compare_hits.m

Código usado com a mesma do utilidade do código anterior mas para os valores de custo médio de procura de um item contido na árvore binária.

```
function compare_hits(file_name)
   file = fopen(file_name, "r");
   data = fscanf(file, "%d %f %f\n");

   max_nodes_per_level = zeros(1, 14);
   max_nodes_per_level(1) = 1;
   max_nodes_per_level(2) = 2;

   for i = 3:length(max_nodes_per_level)
           max_nodes_per_level(i) = max_nodes_per_level(i - 1)*2;
   end

   ns = data(1:3:end);
   cs = cumsum(max_nodes_per_level);
   hit_theor_min = zeros(length(ns), 1);
   hit_theor_max = zeros(length(ns), 1);
   for i = 1:length(hit_theor_min)
           cs_max = cumsum([1:ns(i)]);
           hit_theor_max(i) = cs_max(end);
           for n = 1:length(max_nodes_per_level)
           if  ns(i) <= cs(n)
           hit_theor_min(i) = hit_theor_min(i) + (ns(i) - cs(n - 1))*n;
           break;
           end
           hit_theor_min(i) = hit_theor_min(i) + max_nodes_per_level(n)*n;
           end
   end

   plot(ns(1:end), data(2:3:end), '.', 'MarkerSize', 30);
   hold on;
   plot(ns(1:end), hit_theor_min./ns, '.-', 'LineWidth', 3, 'MarkerSize', 30);
   hold on;
   plot(ns(1:7), hit_theor_max(1:7)./ns(1:7), '.-', 'LineWidth', 3, 'MarkerSize', 30);
   title("Pratical vs theorerical number of calls on hit", 'FontSize', 22);
   xlabel("Number of nodes (n)", 'FontSize', 20);
   ylabel("Number of calls on hit", 'FontSize', 20);
   legend({"Pratical result", "Theoretical expectation of min number of calls", "Theoretical expectation of max number of calls"}, 'FontSize', 20, 'Location', 'southeast');
   set(gca, 'FontSize', 20);
end
```

4.5. compare_miss.m

Código usado com a mesma do utilidade do código “compare_heights.m” mas para os valores de custo médio de procura de um item de informação não pertence à árvore.

```
function compare_miss(file_name)
   file = fopen(file_name, "r");
   data = fscanf(file, "%d %f %f\n");

   max_nodes_per_level = zeros(1, 14);
   max_nodes_per_level(1) = 1;
   max_nodes_per_level(2) = 2;

   for i = 3:length(max_nodes_per_level)
           max_nodes_per_level(i) = max_nodes_per_level(i - 1)*2;
   end

   ns = data(1:3:end);
   cs = cumsum(max_nodes_per_level);
   miss_theor_min = zeros(length(ns), 1);
   miss_theor_max = zeros(length(ns), 1);
   for i = 1:length(miss_theor_min)
           cs_max = cumsum([1:ns(i)]);
           miss_theor_max(i) = cs_max(end);
           for n = 1:length(max_nodes_per_level)
           if  ns(i) <= cs(n)
           miss_theor_min(i) = (max_nodes_per_level(n) - ns(i) + cs(n - 1))*n;
           miss_theor_min(i) = miss_theor_min(i) + (ns(i) - cs(n - 1))*2*(n + 1);
           break;
           end
           end
   end

   plot(ns(1:end), data(2:3:end), '.', 'MarkerSize', 30);
   hold on;
   plot(ns(1:end), miss_theor_min(1:end)./ns(1:end), '.-', 'LineWidth', 3, 'MarkerSize', 30);
   hold on;
   plot(ns(1:7), miss_theor_max(1:7)./ns(1:7), '.-', 'LineWidth', 3, 'MarkerSize', 30);
   title("Pratical vs theorerical number of calls on miss", 'FontSize', 22);
   xlabel("Number of nodes (n)", 'FontSize', 20);
   ylabel("Number of calls on hit", 'FontSize', 20);
   legend({"Pratical result", "Theoretical expectation of min number of calls", "Theoretical expectation of max number of calls"}, 'FontSize', 20, 'Location', 'southeast');
   set(gca, 'FontSize', 20);
end
```

4.6. stg_g.m

Este código é responsável por ilustrar os gráficos do desvio padrão e de os converter para imagem “.png”.

```
clear all;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Este script gera imagens png dos graficos
%% o nome dos ficheiros tem que ser iguais aos nomes do array output
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

output=["count_hit_data","count_miss_data","height_data","leaves_data"];
titles=["Standard deviation of number of calls on hit","Standard deviation of number of calls on miss","Standard deviation of maximum tree height ","Standard deviation of number of leaves"];
ylabels=["std of calls on hit", "stf of calls on miss", "std of maximum height", "std of number of leaves"];
height_and_leaves="%d %d %d %f %f\n";
counts_hit_miss="%d %f %f\n";
format_type=counts_hit_miss;
divisor=3;
for i=1:length(output)
          file_name="";
          file_name=strcat(output(i),'.txt');
          file=fopen(file_name,"r");
          if (i>2)
          format_type=height_and_leaves;
          divisor=5;
          end
          A=fscanf(file,format_type);
          N=length(A);
          std=zeros(N/divisor,1);
          n_values=zeros(N/divisor,1);
          k=1;
          for j=1:divisor:N
          n_values(k)=A(j);
          std(k)=A(j+2+rem(divisor,3));
          k=k+1;
          end
          figure('Position',[0 0 1920 1080]); %change this to set resolution of graph
          hold on;
          grid on;
          plot(n_values,std,'LineWidth', 3, 'MarkerSize', 30);
          title(titles(i), 'FontSize', 22);
          xlabel("Number of nodes (n)", 'FontSize', 20);
          ylabel(ylabels(i), 'FontSize', 20);
          set(gca, 'FontSize', 20);
          output(i)=strcat(output(i),"_std");
          set(gcf,'PaperPositionMode','auto')
          print(output(i),'-dpng','-r0')
          fclose(file);
end
```