Relatório EP2 - MAC0122 2015

Luca Serafini
No. USP 9373434

Lucas Santos
No. USP 9345064

Mardem Junior
No. USP 9065976

7 de Outubro de 2015

1 Introdução

O objetivo desse segundo exercício-programa (EP) consiste em desenhar uma árvore de decisão de diversos algoritmos de ordenação por comparação. Para isso, será utilizado a biblioteca gráfica desenvolvida no primeiro exercício-programa.

Os algoritmos que serão trabalhos nesse EP serão: selection sort, bubblesort, insertion sort, mergesort e heapsort. Adicionalmente, serão incluídas três implementações do quicksort: a primeira será da biblioteca padrão (stdlib), a segunda será uma implementação com aleatorização e a terceira sem aleatorização.

Nesse presente relatório explicaremos a solução desenvolvida, todas as complicações, otimizações e por fim, apresentaremos o resultado final, com desenhos e um manual de como usar o programa.

2 Algoritmos

Abaixo descreveremos como cada algoritmo funciona, apresentaremos uma implementação em linguagem C e, por fim, seus respectivos tempos de execução (no pior, melhor e médio casos).

Na descrição dos algoritmos abaixo admitiremos que possuímos um vetor de inteiros V com N elementos. O objetivo será, portanto, ordenar o vetor V.

Assumiremos ainda que a implementação da função swap existe e seu funcionamento é realizar um intercâmbio dos valores de duas variáveis. Vamos

utilizar as funções de comparação abaixo que serão exploradas no decorrer deste relatório:

greater(int a, int b): retorna 1 se a é maior que b, 0 do contrário;

less(int a, int b): retorna 1 se a é menor que b, 0 do contrário;

2.1 Selection sort

2.1.1 Como funciona

Esse algoritmo talvez seja o algoritmo mais ingênuo e fácil de ser entendido de todos que serão apresentados. Seu funcionamento é direto: seja i inicialmente igual a 0, itera-se o intervalo [i, N), buscando o menor elemento de índice u. Trocamos o elemento i com u e incrementamos i. Repetimos esse processo enquanto i for menor que N.

2.1.2 Implementação usual em C

```
1   void selectionSort(int *v, int N) {
2     int i, j, u;
3
4     for (i = 0; i < N; i++) {
5       u = i;
6       for (j = i + 1; j < N; j++) {
7         if (greater(v[u], v[j]))
8           u = j;
9       }
10
11      swap(&v[u], &v[i]);
12    }
13  }
```

2.1.3 Tempo de execução

No pior caso: $O\left(n^2\right)$
No melhor caso: $O\left(n^2\right)$
No caso médio: $O\left(n^2\right)$

2.2 Bubble sort

2.2.1 Como funciona

O funcionamento desse algoritmo é também bem simples. Seja j = N-1, iteramos i no intervalo [0, j) comparando o i-ésimo elemento com o (i + 1)-ésimo elemento. Caso o i-ésimo elemento seja menor que seu sucessor, trocamos eles de posição e incrementamos i. Quando a iteração chegar ao fim, decrementamos j e repetimos o processo enquanto j for maior que 0. Uma implementação mais otimizada verifica se houve alguma troca. Caso não haja, significa que o vetor está ordenado e podemos parar. Essa otimização torna o tempo de execução do algoritmo em $O(n)$ no melhor caso.

2.2.2 Implementação usual em C

```
1   void bubbleSort(int *v, int N) {
2     int i, j, changed;
3
4     for (j = N-1; j > 0; j -= 1) {
5       changed = 0;
6
7       for (i = 0; i < j; i++) {
8         if (greater(v[i], v[i+1])) {
9           swap(&v[i], &v[i+1]);
10          changed = 1;
11        }
12      }
13
14      if (changed == 0) break;
15    }
16  }
```

2.2.3 Tempo de execução

No pior caso: $O(n^2)$

No melhor caso: $O(n)$ quando otimizado

No caso médio: $O(n^2)$

2.3 Insertion sort

2.3.1 Como funciona

O mais eficiente dos algoritmos de ordenação com implementação mais simples (bubblesort e selection sort), seu funcionamento consiste em manter o inicio do vetor ordenado e a cada passo, insere um novo elemento na posição correta no intervalo já ordenado.

2.3.2 Implementação usual em C

```
1   void insertionSort(int *v, int N) {
2     int i, j, key;
3
4     for (i = 1; i < N; i++) {
5       key = v[i];
6       j = i-1;
7       while(j >= 0 && greater(v[j], key)) {
8         v[j+1] = v[j];
9         j -= 1;
10      }
11      v[j+1] = key;
12    }
13  }
```

2.3.3 Tempo de execução

No pior caso: $\mathrm{O}\left(n^2\right)$
No melhor caso: $\mathrm{O}\left(n\right)$
No caso médio: $\mathrm{O}\left(n^2\right)$

2.4 Mergesort

2.4.1 Como funciona

O algoritmo Mergesort utiliza a estratégia "dividir para conquistar". Através de código recursivo, um vetor é dividido em várias seções menores, que, após serem ordenadas, são intercaladas, utilizando-se de um vetor auxiliar, de forma que o vetor todo estará em ordem crescente após a execução do algoritmo.

O algoritmo utiliza como base da recursão o fato de um vetor com um elemento já está ordenado.

2.4.2 Implementação usual em C

```
1   void merge(int *v, int p, int q, int r) {
2     int i, j, k, *aux;
3     aux = malloc((r - p) * sizeof(int));
4     i = p;
5     j = q;
6     k = 0;
7
8     while (i < q && j < r) {
9       if (greater(v[i], v[j])) aux[k++] = v[j++];
10      else aux[k++] = v[i++];
11    }
12
13    while (i < q) aux[k++] = v[i++];
14    while (j < r) aux[k++] = v[j++];
15
16    for (i = p; i < r; i++) {
17      v[i] = aux[i-p];
18    }
19
20    free(aux);
21  }
22
23  void mergesort(int *v, int p, int q) {
24    if (p < q - 1) {
25      int r = (p + q) / 2;
26      ep_mergesort(v, p, r);
27      ep_mergesort(v, r+1, q);
28      merge(v, p, r, q);
29    }
30  }
```

2.4.3 Tempo de execução

No pior caso: $\mathrm{O}\Big(n \log n\Big)$
No melhor caso: $\mathrm{O}\Big(n \log n\Big)$
No caso médio: $\mathrm{O}\Big(n \log n\Big)$

2.5 Heapsort

2.5.1 Como funciona

O Heapsort utiliza a estrutura de dados chamada *heap binário*, que fica escondida dentro dos vetores. Um (max) heap é um vetor v[1..m] tal que

$$\mathrm{v[f]} \geq \mathrm{v[f/2]}$$

para f = 2, ..., m. Vale ressaltar que f/2 deve ser entendido como $\lfloor f/2 \rfloor$. Nosso objetivo, nesse algoritmo, passa a ser transformar o vetor em um *heap*.

Para isso, precisamos de uma função peneira que desça um elemento até sua posição correta. Somado a essa função, utilizamos uma segunda função, *heapsort*, que desce, caso necessário, todos os elementos para as posições corretas.

2.5.2 Implementação usual em C

```
1   void peneira(int *v, int p, int m) {
2     int f = 2*p, x = v[p];
3
4     while (f <= m) {
5       if (f < m && less(v[f], v[f+1]))
6         f++;
7
8       if (x >= v[f])
9         break;
10
11      v[p] = v[f];
12      p = f;
13      f = 2*p;
14    }
15
16    v[p] = x;
17  }
18
19  void heapsort(int *v, int n) {
20     int p, m;
21
22    for (p = n/2; p >= 1; p--)
23      peneira(v, p, n);
24
25    for (m = n; m >= 2; m--) {
26      swap(&v[1], &v[m]);
27      peneira(v, 1, m-1);
28    }
29  }
```

2.5.3 Tempo de execução

No pior caso: $\mathrm{O}\big(n \log n\big)$

No melhor caso: $\mathrm{O}\big(n \log n\big)$

No caso médio: $\mathrm{O}\big(n \log n\big)$

2.6 Quicksort (biblioteca padrão)

O funcionamento assemelha-se ao do quicksort sem aleatorização, descrito a seguir, mas sua implementação exata tal como seu tempo de execução não é especificado. Descreveremos a implementação da função de comparação que é utilizada no qsort da biblioteca padrão:

```
1  int qsort_compare_func(const void * a, const void *
       b) {
2    int res = less(*(int *)a, *(int *)b);
3    return res ? -1 : 1;
4  }
```

Como estamos trabalhando com um vetor de elementos onde não temos repetição de elementos, é seguro retornar apenas se um elemento é maior ou menor que o outro, ignorando o caso de igualdade.

2.7 Quicksort sem aleatorização

2.7.1 Como funciona

O Quicksort utiliza a estratégia de "dividir para conquistar". A formulação do algoritmo nas palavras do professor Paulo Feofiloff, do IME-USP:

"rearranjar um vetor v[p..r] de modo que todos os elementos pequenos fiquem na parte esquerda do vetor e todos os elementos grandes fiquem na parte direita."

Essa formulação é feita recursivamente tendo como base o vetor nulo, que já está ordenado.

2.7.2 Implementação usual em C

```
1   int separar(int *v, int p, int r) {
2     int x = v[r],
3       i = p - 1,
4       j = 0;
5
6     for (j = p; j < r; j++) {
7       if (less(v[j], x)) {
8         i++;
9         swap(&v[i], &v[j]);
10      }
11    }
12
13    swap(&v[i+1], &v[r]);
14
15    return i + 1;
16  }
17
18  void quicksort(int *v, int p, int r, int random) {
19    if (p < r) {
20      int q = random == 1 ? separar_random(v, p, r) :
            separar(v, p, r);
21      quicksort(v, p, q - 1, random);
22      quicksort(v, q + 1, r, random);
23    }
24  }
```

2.7.3 Tempo de execução

No pior caso: $\mathrm{O}\left(n^2\right)$

No melhor caso: $\mathrm{O}\left(n \log n\right)$

No caso médio: $\mathrm{O}\left(n \log n\right)$

2.8 Quicksort com aleatorização

2.8.1 Como funciona

Como o nome já diz, o Quicksort é um algoritmo de rápida ordenação, sendo considerado um dos mais eficientes. Porém, sua eficiência é da ordem

de $O\left(n^2\right)$ no pior caso, ou seja, quando o vetor está praticamente ordenado e o pivô é o menor ou maior número. Isso faz com que tal algoritmo seja comparado aos algoritmos elementares nesses casos raros. Para consertar o problema, o pivô pode ser escolhido aleatoriamente, através da função rand da biblioteca *"stdlib.h"*, fazendo com que sua eficiência seja da ordem de $O\left(n \log n\right)$ no pior caso.

2.8.2 Implementação usual em C

```
1  int separar_random(int *v, int p, int r) {
2    int random = rand() % (r - p) + p;
3    swap(&v[r], &v[random]);
4    return separar(v, p, r);
5  }
```

O resto da implementação é exatamente igual ao do Quicksort sem aleatorização.

2.8.3 Tempo de execução

No pior caso: $O\left(n \log n\right)$
No melhor caso: $O\left(n \log n\right)$
No caso médio: $O\left(n \log n\right)$

3 Solução

Para construimos a solução final, isto é, gerar o arquivo de imagem da árvore, precisamos quebrar o problema em problemas menores. O funcionamento do programa funciona em algumas etapas:

- **Implementação dos algoritmos de ordenação:** os algoritmos de ordenação devem ser escritos corretamente (a implementação, contudo, já foi descrita no capítulo anterior);
- **Permutar todas as combinações possíveis:** Dado um N de entrada, precisamos gerar todas as permutações de um vetor com N elementos distintos;
- **Analisar o comportamento do algoritmo (geração da árvore de decisão):** dado um algoritmo inserido na entrada do programa,

para cada permutação, precisamos ver como o algoritmo se comporta (quais decisões ele faz). Com isso, conseguimos construir uma árvore binária com as decisões do algoritmo escolhido;

- **Calcular as posições dos nós e arestas:** dado a árvore binária, precisamos de um algoritmo que calcule a posição no plano cartesiano de cada nó a ser inserido no desenho;
- **Gerar o desenho:** conhecendo as posições, usamos a biblioteca desenvolvida no exercício-programa 1 para criar nosso desenho (Drawing);
- **Salvar o arquivo de imagem (PGM):** possuindo o desenho (Drawing), podemos salva-lo em um arquivo de imagem;
- **Fim**;

3.1 Implementação dos algoritmos de ordenação

A implementação dos algoritmos já foram descritas no capítulo anterior.

3.2 Permutar todas as combinações possíveis

Dado um valor N, queremos saber quais são todas as permutações possíveis de um vetor com N elementos distintos. Por simplificação, os elementos do vetor serão de 1 à N (1, 2, ..., N).

A utilidade de conhecermos todas as permutações na construção do nosso exercício-programa se resume a conhecer quais decisões são feitas pelo algoritmo escolhido para todas as possíveis entradas. Conhecendo todas as decisões de todas as permutação, somos capazes de determinar a árvore de decisões.

Abaixo, temos uma implementação recursiva para gerar as permutações:

```
1   void processEachPermutation(int n, int *v, int
       index) {
2     if (n == index) {
3       /* Aqui, o vetor v é uma permutação válida.
           Podemos trabalhar com ele. */
4
5     } else for (int j = index; j < n; j++) {
6       swap(&elements[index], &elements[j]);
7       processEachPermutation(n, v, index + 1);
8       swap(&elements[index], &elements[j]);
9
10    }
11  }
```

O tempo de execução desse algoritmo é $O(n!)$. O fato do tempo de execução do algoritmo crescer com base no fatorial, torna essa etapa muito lenta. E não é muito difícil entendermos isso: para N = 8, temos que 8! = 40.320 e para N = 9, temos que 9! = 362.880. Ao passo que N aumentou em apenas uma unidade, N! aumentou em 322.560.

No capítulo de **Otimizações** discutiremos o tempo de execução desse método, tal como uma implementação iterativa e uma implementação onde pré-calculamos e armazenamos todas as permutações possíveis.

3.3 Analisar o comportamento do algoritmo (geração da árvore de decisão)

Agora que possuímos os algoritmos implementados e as permutações disponíveis, precisamos analisar e armazenar como cada permutação se comporta no algoritmo.

No capítulo de implementação dos algoritmos, esboçamos quatro funções customizadas de comparação: “greater” e “less”. A vantagem de usarmos operadores customizadas aparece aqui. Utilizando operadores customizados podemos ir construir a árvore diretamente ao usar esses operadores.

Seja “nó” uma estrutura duas variáveis: “esquerda” (do tipo “nó”) e “direita"(do tipo “nó”).

No início do programa definimos a raiz da árvore como um nó e no início do algoritmo de ordenação definimos o nó atual apontando para a raiz da árvore. A cada comparação entre dois valores, “a” e “b”, temos duas possibilidades:

- Ir para a esquerda, caso “a” seja menor que “b”. Nesse caso, definimos o nó atual como o nó à esquerda do nó atual (caso não esteja alocado na memória, alocamos-o);
- Ir para a direita, caso contrário. Nesse caso, definimos o nó atual como o nó à direita do nó atual (caso não esteja alocado na memória, alocamos-o);

A implementação dos métodos comparativos customizados nos ajuda a fatorar o código que implementa a ideia acima. Isso acontece porque todos os algoritmos que estamos trabalhando são algoritmos de ordenação por comparação e, inevitavelmente, serão feitos comparações entre dois valores em algum momento da execução dos algoritmos.

A implementação dos comparadores é feita da seguinte forma:

```
1   Node **curNode; /* variável global usada para
       apontar o nó atual */
2
3   int less(int a, int b) {
4     if (a < b) {
5       if (!((*curNode)->left))
6         (*curNode)->left = createNode();
7
8       *curNode = (*curNode)->left;
9     } else {
10      if (!((*curNode)->right))
11        (*curNode)->right = createNode();
12
13      *curNode = (*curNode)->right;
14    }
15    return a < b;
16  }
17
18  int greater(int a, int b) {
19    if (a < b) {
20      if (!((*curNode)->left))
21        (*curNode)->left = createNode();
22
23      *curNode = (*curNode)->left;
24    } else {
25      if (!((*curNode)->right))
26        (*curNode)->right = createNode();
27
28      *curNode = (*curNode)->right;
29    }
30    return a > b;
31  }
```

Um ponto interessante é o fato da variável "curNode" (que representa o nó atual) ser do tipo "Node **". O uso do ponteiro duplo acontece pelo fato de estarmos alterando o valor para qual ponteiro ele aponta.

Outro ponto interessante que vale ressaltar é que o tempo de execução dessa etapa é exatamente igual ao do algoritmo escolhido, uma vez que fizemos apenas uma pequena alteração para rastrear as decisões que ele faz.

3.4 Calcular as posições dos nós e arestas

O objetivo dessa etapa é calcular as larguras de cada árvore.

Aqui poderíamos gerar a imagem de uma árvore perfeitamente simétrica, omitindo os nós que não aparecem na árvore de decisão. Contudo isso não seria uma boa solução, uma vez que a largura da imagem ficaria bem maior. Deixar a largura maior do que o necessário não é interessante, pois ocuparia mais espaço na memória e, principalmente, porque o tempo de execução do programa fica maior. Isso acontece porque os segmentos de retas (arestas) que ligam dois nós (vértices) possuiriam um comprimento maior e o tempo de execução para escreve-los na matriz aumentaria. Vale notar que a matriz também deverá ser maior.

Dado que cada nó pode ser visto como uma raiz e, dessa forma, formando uma árvore, uma abordagem interessante para resolver esse problema é uma função recursiva. Dessa forma, a largura de uma árvore é dada pela soma das larguras das árvores à esquerda e à direita. E caso o nó seja uma folha, isto é, não apresentar nós à esquerda e à direita, então temos que sua largura é a largura de um nó (definimos essa condição como base da nossa recursão).

Temos que a implementação da função recursiva é dada por:

```
1   void calculateWidth(Node *node, double scale_x,
       double thickness) {
2     if (!node->left && !node->right) {
3       node->width = SPACE_PER_NODE_X(thickness,
           scale_x);
4
5     } else {
6       double width = SPACE_HORIZONTAL;
7
8       if (node->left) {
9         calculateWidth(node->left, scale_x, thickness);
10        width += node->left->width;
11      }
12
13      if (node->right) {
14        calculateWidth(node->right, scale_x,
             thickness);
15        width += node->right->width;
16      }
17
18      node->width = width;
19    }
20  }
```

Observação: foi adicionado a estrutura "Node" (nó) o atributo "width" do tipo *double* para que fosse possível armazenarmos a sua largura.

3.5 Gerar o desenho

Calculado o quanto espaço cada nó ocupa, agora precisamos definir as coordenadas cartesianas. O desafio aqui é saber quanto espaço foi ocupado pelos nós a esquerda para então ocuparmos o próximo espaço vazio.

Novamente, usaremos a mesma abordagem recursiva. A base da recursão é quando o nó não existe e portanto não precisamos inseri-lo.

```
1   void insertNode(Drawing d, Node *node, double
       offsetY, double offsetX, double scale_y, double
       thickness) {
2     if (!node) return;
3
4     add_circle(d, offsetX + node->width / 2.0,
         offsetY, RADIUS(thickness), thickness, 1);
5     node->x = offsetX + node->width / 2.0;
6     node->y = offsetY;
7
8     if (node->left) {
9       insertNode(d, node->left, offsetY +
           SPACE_PER_NODE_Y(scale_y), offsetX, scale_y,
           thickness);
10      add_line_segment(d, node->x, node->y,
           node->left->x, node->left->y, thickness);
11    }
12
13    if (node->right) {
14      insertNode(d, node->right, offsetY +
           SPACE_PER_NODE_Y(scale_y), offsetX +
           nodeWidth(node->left), scale_y, thickness);
15      add_line_segment(d, node->x, node->y,
           node->right->x, node->right->y, thickness);
16    }
17  }
```

O funcionamento desse algoritmo é bem simples: sabendo a largura das árvores filhas, tanto a da esquerda quanto a da direita, insere-se elas (recursivamente) passando como argumento da função recursiva o *offset* horizontal e vertical. Sabendo os *offsets*, sabemos a posição a ser desenhada.

3.6 Salvar o arquivo de imagem (PGM)

Com o desenho (drawing) gerado possuindo todos os círculos (que representam os nós) e segmentos de retas (que representam as arestas), nós podemos simplesmente chamar a função "save_pgm", que foi desenvolvida no exercício-programa 1, passando como argumento o desenho gerado.

3.7 Fim

O programa, nesse ponto, funciona corretamente. Agora no próximo capítulo vamos discutir otimizações no programa afim de executa-lo em um tempo menor.

4 Otimização

4.1 Permutar todas as combinações possíveis

No capítulo anterior, especificamente na etapa 2, definimos um algoritmo recursivo para gerar todas as permutações de um vetor de N elementos. Na linguagem C, sabe-se que na maioria dos casos as funções recursivas são mais lentas que as funções iterativas.

Uma possível otimização no código poderia ser feito, trocando o algoritmo recursivo de permutação por um iterativo. Pesquisando sobre algoritmos iterativos que computam todas as permutações de um dado vetor, encontramos um algoritmo chamado "Counting QuickPerm" que possuí a seguinte implementação:

```
1   void processEachPermutation(int n, int *v, int *p) {
2       int i, j, tmp;
3
4       i = 1;
5       while(i < n) {
6           if (p[i] < i) {
7               j = i % 2 * p[i];
8               swap(&v[j], &v[i]);
9               p[i]++;
10              i = 1;
11          } else {
12              p[i] = 0;
13              i++;
14          }
15      }
16  }
```

Uma terceira possível implementação surge do fato do valor de N estar restrito dentre 2 e 9. Essa informação nos permite pré-calcular e memorizar todas as permutações, colocando-as direto na memória. O trade-off nesse caso seria o custo que teríamos de armazenar todas as permutações na me-

mória. A quantidade de número inteiros que teríamos que armazenar podem ser calculados como o somatório de todas as possibilidades possíveis para todos os possíveis valores de N:

$$\sum_{n=2}^{9} n! = 409.112$$

Considerando que cada inteiro ocupe 4 bytes, concluímos que os 409.112 inteiros ocupariam 1.636.448 bytes, equivalente a 1.63mb. Essa quantidade de memória consumida não se tornaria um problema no caso desse exercício-problema. Essa alternativa torna-se, portanto, uma alternativa viável que avaliaremos a seguir.

Desenvolvemos três programas separados, um com o algoritmo recursivo, outro com o algoritmo iterativo e um terceiro usando memorização afim de comparar o tempo de execução, na prática, dos mesmos. Usando o comando "time" do terminal de um único computador, cronometramos o tempo de execução dos três programas cujo a finalidade é iterar todas as combinações possíveis. O resultado pode ser visto abaixo:

	Algoritmo Recursivo	**Algoritmo Iterativo**	**Memorização**[1]
N = 8	0m0.005s	0m0.004s	0m0.002s
N = 9	0m0.013s	0m0.009s	0m0.004s
N = 10	0m0.122s	0m0.039s	0m0.011s
N = 11	0m1.128s	0m0.364s	0m0.088s
N = 12	0m13.586s	0m4.684s	0m1.017s

Claramente o algoritmo recursivo é o menos eficiente, seguido do iterativo e sendoo método de pré-calcular, dentre as opções apresentadas, o melhor método. Entretanto, optamos utilizar o método iterativo por ser suficientemente rápido para o pior caso (quando N for igual a 9).

4.2 Imprimindo a árvore

Conforme foi detalhado no capítulo anterior, especificamente na etapa 4, estudamos a implementação de dois algoritmos para gerar as posições dos nós e das arestas.

O primeiro algoritmo possui uma solução mais simplificada e um tanto quanto ingênua. Sua execução consiste em calcular a altura H máxima da

[1] O tempo calculado é somente de executar um loop N! vezes e desconsideramos o tempo gasto para pré-calcular os valores.

árvore. Sabendo a altura máxima da árvore, sabemos que a base dela possuí no máximo 2^{H-1} nós. Com isso podemos determinar a largura da base. Esse algoritmo gera uma árvore de largura simétrica e isso é inconveniente uma vez que na base da árvore tem menos nós que 2^{H-1} em alguns casos, ocupando, dessa forma, mais espaço horizontalmente do que deveria.

O segundo algoritmo, implementado no capítulo anterior, calcula as larguras recursivamente e dessa forma sabemos precisamente o espaço que estamos usando, evitando desperdícios de espaço que poderiam levar a um tempo maior de execução, uma vez que a imagem a ser gerada seria maior.

4.3 Otimizando os algoritmos de ordenação

Alguns algoritmos, como o bubble sort, possuem implementações mais eficientes que consomem um tempo menor em alguns casos (no caso do bubble sort, a implementação mais eficiente torna o tempo de execução no melhor caso em $\mathrm{O}\big(n\big)$). Nesse exercício-programa, contudo, evitamos buscar qualquer tipo de otimização nos algoritmos de ordenação se limitando às implementações passadas em aula pelo professor. Com isso, garantimos ainda que a árvore de decisão será completa e correta.

4.4 Otimizando a biblioteca *graphics* do EP 1

Certamente uma boa implementação da função save_pgm pode melhorar, e muito, o tempo de execução do programa no geral. Para o exercício-programa 1, usamos uma abordagem que satisfazia as seguintes etapas:

- Iterava a lista de círculos e segmentos de reta;
  - Para cada elemento da lista, definia-se a submatriz retangular para a qual o elemento contribuí na coloração dos pixels;
  - Itera-se a submatriz calculando a cor em relação ao elemento atual;
- No final da iteração a matriz já possui as cores definidas;
- Imprimir a matriz no arquivo PGM e fim;

Esse algoritmo certamente já é bem eficiente uma vez que ele evita percorrer a matriz a inteira sempre que possível. Mas ainda assim a submatriz que é percorrida possui muitos indices que não interferem na imagem, isto é, indices onde a cor permanece inalterada.

Antes, para calcular um segmento, iteravamos a submatriz que continha o segmento. Contudo, descobrimos que a maneira mais otimizada possível

seria percorrer somente os índices que fazem parte diretamente do elemento e que podem alter a cor da imagem. Para calcular os índices relevantes, foi utilizado noções de geometria e trigonometria simples.

Abaixo temos ilustrações que mostram como era feito antes e depois.

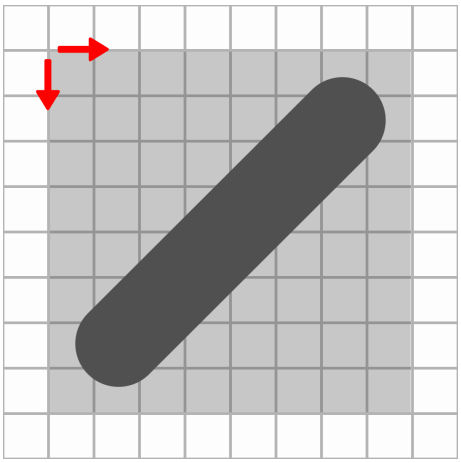

Figura 1: Antes

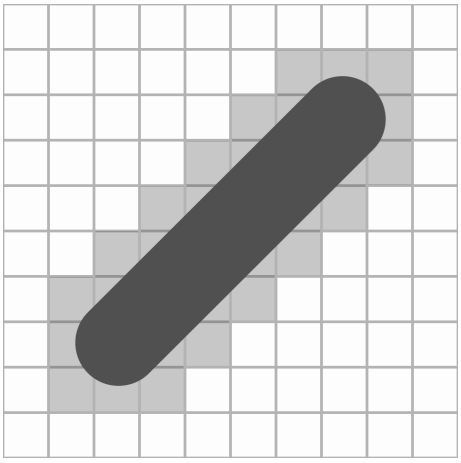

Figura 2: Depois

5 Desenhos

Abaixo temos as imagens geradas pelo exercício-programa desenvolvido, para valores de N = 5 com as seguintes entradas:

Escala X 1.0;

Escala Y 1.0;

Grossura 3.0;

Pixels por unidade 10;

Tamanho da amostra 3;

Tipo de amostra grid;

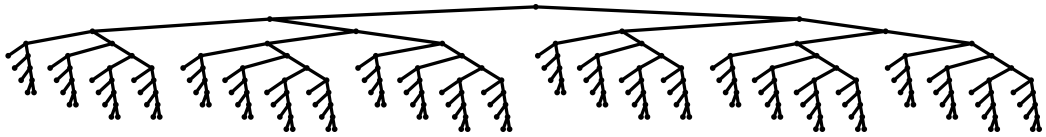

Figura 3: Insertion sort

Figura 4: Bubble sort

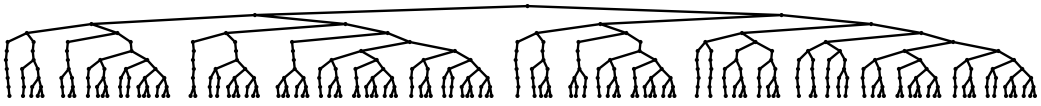

Figura 5: Selection sort

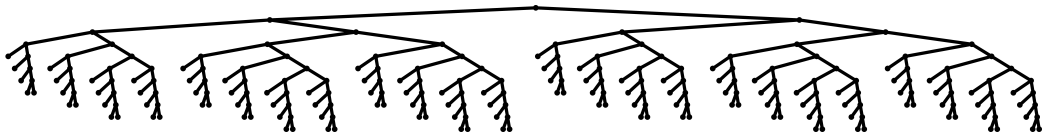

Figura 6: Qsort (biblioteca padrão)

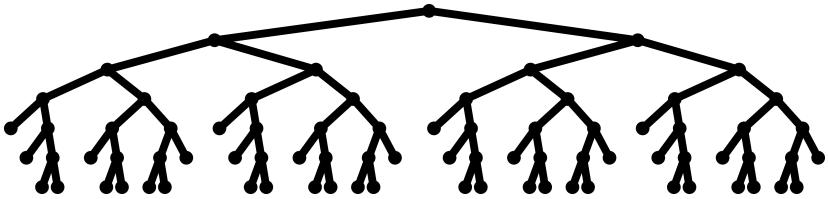

Figura 7: Mergesort

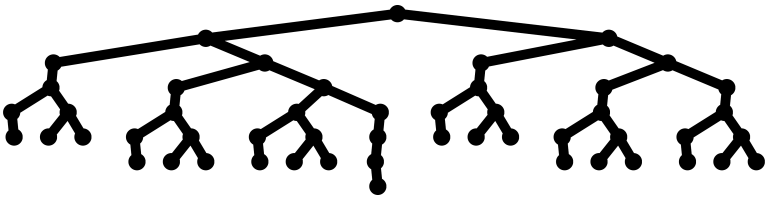

Figura 8: Heapsort

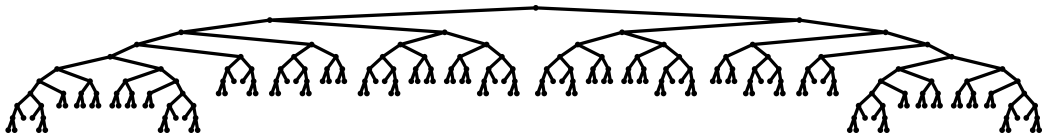

Figura 9: Quicksort sem aleatorização

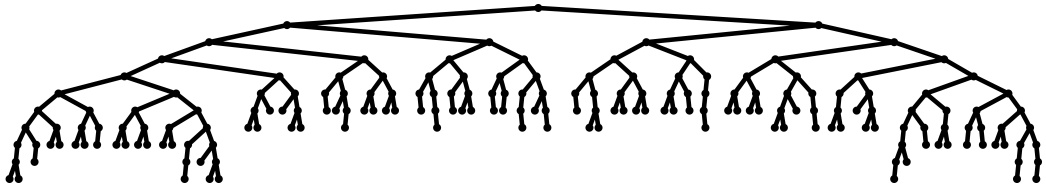

Figura 10: Quicksort com aleatorização

6 Manual de uso

6.1 Estrutura dos arquivos

A estrutura dos arquivos é bem simples. Segue a descrição do que cada arquivo é encarregado de fazer:

1. **builder** (.c, .h): calcular a largura de uma árvore, as posições de cada nó e aresta e, por fim, integrar com a biblioteca *graphics*;
2. **graphics** (.c, .h): biblioteca escrita no EP 1 e otimizada agora, serve para criar desenhos e salva-los no formato *.pgm*;
3. **main** (.c): recebe as entradas pela linha de comando e dá início ao programa;
4. **tree** (.c, .h): responsável por obter uma árvore de um algoritmo;

6.2 Compilando

O exercício-programa conta com um arquivo Makefile que torna todo o processo de compilação mais simples. Para compilar, basta digitar no terminal "*make*". Isso já será o suficiente para compilar todos os arquivos, gerando o executável.

6.3 Como usar

Esse exercício-programa foi feito para funcionar somente pela linha de comando, exatamente como foi feito no exercício-programa anterior. O uso se dá da seguinte forma:

tree <algoritmo> <num_elementos> <outfile> <scale_x> <scale_y> <grossura> <pix_por_unidade> <tamanho_amostra> <tipo_amostra>

- *algoritmo:* qual algoritmo a árvore deve representar. Disponivel: qsort, selection_sort, insertion_sort, bubble_sort, quicksort_random, quicksort, mergesort, heapsort;
- *num_elementos:* (número inteiro) número de elementos para ser sorteado. Tenha cuidado: evite escolher números altos (o programa ainda funcionará, mas seu tempo de execução cresce rapidamente);

- *outfile:* nome do arquivo de saída que a imagem deve ser escrita;
- *scale_x:* (número real) fator escalar na largura (eixo X);
- *scale_y:* (número real) fator escalar na vertical (eixo Y);
- *grossura:* (número real) grossura das linhas e dos nós;
- *pix_por_unidade:* (número inteiro) número de pixels por unidade;
- *tamanho_amostra:* (número inteiro) tamanho da amostra;
- *tipo_amostra:* (opcional) ou "grid", para um gride uniforme, ou "random" para um grid rândomico. Grid por padrão;

A saída do programa se limita à geração do arquivo com nome igual à entrada no programa pelo parâmetro "outfile".