CAPÍTULO 4

Implementações de Filas de Prioridade

O algoritmo de Dijkstra, segundo o teorema 3.5, realiza uma seqüência de n insert, n delete-min e no máximo m decrease-key operações, em um grafo com n vértices e m arcos. Logo, o consumo de tempo do algoritmo de Dijkstra pode variar conforme a implementação de cada uma dessas operações da fila de prioridade.

Existem muitos trabalhos envolvendo implementações de filas de prioridade com o intuito de reduzir o tempo gasto pelo algoritmo de Dijkstra. Para citar alguns bons exemplos temos [3, 7, 16].

As estruturas de dados utilizadas na implementação das filas de prioridade podem ser divididas em duas categorias, conforme as operações elementares utilizadas:

(1) (modelo de comparação) estruturas baseadas em comparações; e

(2) (modelo RAM) estruturas baseadas em "bucketing".

Bucketing é um método de organização dos dados que particiona um conjunto em partes chamadas **buckets**. No que diz respeito ao algoritmo de Dijkstra, cada bucket agrupa vértices de um grafo relacionados através de prioridades, que nesse caso, são os potenciais.

Neste capítulo, são descritas as implementações das estruturas de dados *heap,* D-*heap* e *fibonacci heap* que são baseadas em comparações e as estruturas *bucket heap*[1] e *radix heap* que trabalham no modelo RAM. Nas implementações, cada item da fila será um vértice e a prioridade desse vértice será um valor numérico, o seu potencial. As implementações foram baseadas nos livros, Network Flows [1], Introduction to Algorithms [8], The Stanford GraphBase [27] e no livro de Schrijver [36].

[1]Na literatura, a implementação do algoritmo que utiliza buckets é conhecida como implementação de Dial [10].

4.1 Heap

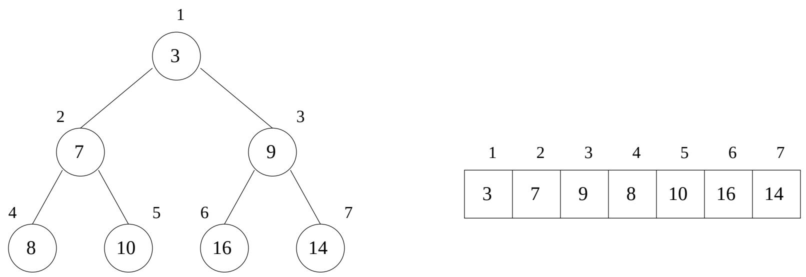
Um **heap** é uma estrutura de dados que mantêm uma seqüência de itens, onde cada item é acessado através de um índice numérico. No nosso caso, os itens são vértices, que serão dispostos em um vetor Q conforme o seu potencial. A organização dos vértices em Q respeita a seguinte propriedade:

Propriedade 4.1 (ordem):

$$d(Q[\lfloor i/2 \rfloor]) \leq d(Q[i]),\ \textit{para cada posição } i \textit{ em } [2..qsize],$$

onde qsize representa o número de vértices em Q.

Intuitivamente, o vetor Q também pode ser visto como uma árvore binária, onde cada posição do vetor representa um vértice da árvore. A figura 4.1 ilustra essa representação.



(a) (b)

Figura 4.1: (a) um heap visto como uma árvore binária; (b) o mesmo heap visto como um vetor.

Na implementação, o vetor Q é simulado utilizando-se o campo utilitário x da estrutura do SGB (seção 1.7), ou seja, $(Q+i)\to x.V$ aponta para o vértice da posição i. Para simplificar a notação, $(Q+i)\to x.V$ é definido como $Q(i)$. Além do mais, se $Q(i)$ aponta para o vértice v, então $v\to posicao = i$.

23 ⟨ Definições 12 ⟩ +≡
#**define** $Q(i)$ $(Q+i)\to x.V$
#**define** *posicao* $w.I$

24 ⟨ Variáveis globais 13 ⟩ +≡
Vertex $*Q$; /* heap */

Para criar o heap Q é utilizada a função *create_heap*.

25 ⟨ Filas de prioridade 11 ⟩ +≡

```
void create_heap(g)     /* cria um heap vazio */
    Graph *g;
{
  register Vertex *v;
  Q = g→vertices;
  for (v = g→vertices; v < g→vertices + g→n; v++) {
    v→posicao = 0;
    v→x.V = Λ;
  }
}
```

As implementações das operações insert, delete-min e decrease-key, são representadas pelas funções *insert_heap*, *delete_min_heap* e *decrease_key_heap*.

26 ⟨ Filas de prioridade 11 ⟩ +≡

```
void insert_heap(Vertex *v);
Vertex *delete_min_heap( );
void decrease_key_heap(Vertex *v);
```

O tempo gasto para inserir um vértice no heap é $O(1)$, somado ao tempo consumido pela operação *decrease_key_heap*. Essa operação funciona como se o último vértice tivesse seu potencial alterado (diminuído), então é necessário o reposicionamento deste vértice em Q, de modo que a propriedade 4.1 continue sendo respeitada.

27 ⟨ Filas de prioridade 11 ⟩ +≡

```
void insert_heap(v)
    Vertex *v;
{
  v→posicao = qsize + 1;
  decrease_key_heap(v);
}
```

Em um heap, o vértice com o menor potencial se encontra na primeira posição, ou seja, em $Q(1)$. Porém, após sua remoção, é preciso encontrar o novo vértice com potencial mínimo. No pior caso, essa operação gasta tempo $O(\log qsize)$.

28 ⟨Filas de prioridade 11⟩ +≡

```
Vertex *delete_min_heap()
{
  register unsigned long p, f;
  register Vertex *v, *vmin;
  if (qsize ≡ 0) return Λ;
  vmin = Q(1);
  v = Q(qsize);
  p = 1;
  f = p ≪ 1;    /* f = 2p */
  while (f ≤ (qsize − 1)) {
    if ((f < (qsize − 1)) ∧ (Q(f)→dist > Q(f + 1)→dist)) f++;
    if (Q(f)→dist ≥ v→dist) break;
    Q(p) = Q(f);
    Q(p)→posicao = p;
    p = f;
    f = p ≪ 1;
  }
  Q(p) = v;
  Q(p)→posicao = p;
  return vmin;
}
```

Diminuir o potencial de um vértice, ao visitá-lo, pode tornar necessário reposicioná-lo em Q, de modo que a propriedade 4.1 continue sendo respeitada. No pior caso, essa operação gasta tempo $O(\log qsize)$.

29 ⟨Filas de prioridade 11⟩ +≡

```
void decrease_key_heap(v)
    Vertex *v;
{
  register unsigned long p, f;
  f = v→posicao;
  p = f ≫ 1;    /* ⌊f/2⌋ */
  while ((p > 0) ∧ (Q(p)→dist > v→dist)) {
    Q(f) = Q(p);
    Q(f)→posicao = f;
    f = p;
```

```
    p = f ≫ 1;
  }
  Q(f) = v;
  Q(f)→posicao = f;
}
```

Lema 4.2 (operações heap): *Na implementação da fila de prioridade Q, utilizando um heap, cada operação insert, delete-min e decrease-key, gasta tempo $O(\log n)$.* ∎

Portanto, do teorema 3.5 e do lema 4.2, o tempo gasto pelo algoritmo de Dijkstra utilizando um heap, é

$$\underbrace{O(n \log n)}_{\text{insert}} + \underbrace{O(m \log n)}_{\text{decrease-key}} + \underbrace{O(n \log n)}_{\text{delete-min}} = O(m \log n).$$

Teorema 4.3: *A implementação do algoritmo de Dijkstra que utiliza um heap resolve o problema do caminho mínimo em um grafo com n vértices e m arcos em tempo $O(m \log n)$.* ∎

4.2 D-heap

Um **D-heap**, assim como o heap (seção 4.1), é uma estrutura de dados que mantêm uma seqüência de vértices, num vetor Q, dispostos de maneira organizada em relação a função potencial, de maneira que:

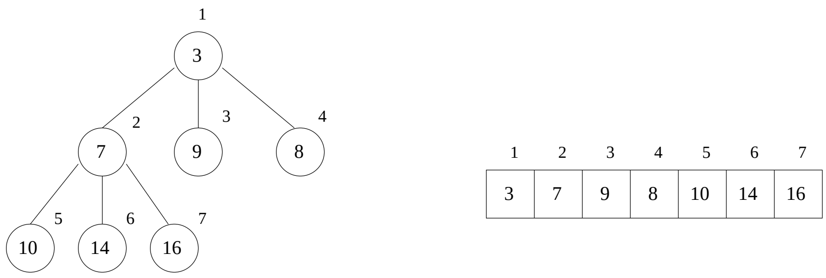
Propriedade 4.4 (ordem):

$$d(Q[\lceil (i-1)/\mathrm{D} \rceil]) \leq d(Q[i]), \text{ para cada posição } i \text{ em } [2..qsize],$$

onde $qsize$ representa o número de vértices em Q e D *é o maior número de filhos de um vértice.*

Intuitivamente, o vetor Q pode ser visto como uma árvore, em que cada vértice pertencente a ele pode ter no máximo D filhos. A figura 4.2 ilustra a representação de um 3-heap.

Na implementação, D é o maior número de filhos de um vértice. A escolha de D é feita dinamicamente, em relação a $\max\{2, \lceil m/n \rceil\}$. O motivo pelo qual D é escolhido dessa maneira



(a) (b)

Figura 4.2: Ilustração de um 3-heap. Em (a) o 3-heap é visto como uma árvore e em (b) como um vetor.

é mostrado no final da seção.

31 ⟨ Variáveis globais 13 ⟩ +≡

```
unsigned long D;      /* número máximo de filhos de um vértice */
```

A função *teto*(i, j) devolve $\lceil i/j \rceil$. A primeira utilidade da função é encontrar um valor para D.

32 ⟨ Funções auxiliares 32 ⟩ ≡

```
unsigned long teto(i, j)
    unsigned long i;
    unsigned long j;
{
  return (i % j ≡ 0 ? i/j : i/j + 1);
}
```

Veja também blocos 41, 50, 60, 73, 74, 75, 86, 96, 105 e 129.

Este código é usado no bloco 9.

Para criar o D-heap Q é utilizada a função *create_dheap*.

33 ⟨ Filas de prioridade 11 ⟩ +≡

```
void create_dheap(g)      /* cria um heap vazio */
    Graph *g;
{
  unsigned long t;
  register Vertex *v;
```

```
Q = g->vertices;
for (v = g->vertices; v < g->vertices + g->n; v++) {
   v->posicao = 0;
}
t = teto(g->m/2, g->n);      /* número de arestas é o dobro do número de arcos */
D = t > 2 ? t : 2;
}
```

As implementações das operações insert, delete-min e decrease-key, são representadas pelas funções *insert_dheap*, *delete_min_dheap* e *decrease_key_dheap*.

34 ⟨ Filas de prioridade 11 ⟩ +≡

```
void insert_dheap(Vertex *v);
Vertex *delete_min_dheap();
void decrease_key_dheap(Vertex *v);
```

Inserir um vértice no D-heap funciona de forma análoga a inserir um vértice no heap.

35 ⟨ Filas de prioridade 11 ⟩ +≡

```
void insert_dheap(v)
      Vertex *v;
{
   v->posicao = qsize + 1;
   decrease_key_dheap(v);
}
```

Assim como no heap 4.1, o vértice com o menor potencial pode ser encontrado na primeira posição, ou seja, em $Q(1)$. E após sua remoção, é preciso encontrar o novo vértice com potencial mínimo. No pior caso, essa operação gasta tempo $O(D \log_D qsize)$

36 ⟨ Filas de prioridade 11 ⟩ +≡

```
Vertex *delete_min_dheap()
{
   register unsigned long p, f, i, ultimo_filho;
   register Vertex *v, *vmin;
   if (qsize ≡ 0) return Λ;
   vmin = Q(1);
   v = Q(qsize);
```

```
    p = 1;      /* os filhos do vértice da posicão p, são encontrados nas posições
          pD − D + 2, . . . , min{n, pD + 1} */
    f = p * D − D + 2;
    while (f ≤ (qsize − 1)) {
      ⟨ Encontra filho f de p tal que d(f) seja mínimo 37 ⟩
      if (Q(f)→dist ≥ v→dist) break;
      Q(p) = Q(f);
      Q(p)→posicao = p;
      p = f;
      f = p * D − D + 2;
    }
    Q(p) = v;
    Q(p)→posicao = p;
    return vmin;
  }
```

Encontra o $\min\{Q(f)\to dist\}$ para cada filho f de p.

37 ⟨ Encontra filho f de p tal que $d(f)$ seja mínimo 37 ⟩ ≡

```
  ultimo_filho = (qsize − 1) < p * D + 1 ? (qsize − 1) : p * D + 1;
  for (i = f + 1; i ≤ ultimo_filho; i++)
    if (Q(f)→dist > Q(i)→dist) f = i;
```

Este código é usado no bloco 36.

Diminuir o potencial de um vértice, ao visitá-lo, pode tornar necessário reposicioná-lo em Q, de modo que a propriedade 4.4 continue sendo respeitada. No pior caso, essa operação gasta tempo $O(\log_D qsize)$.

38 ⟨ Filas de prioridade 11 ⟩ +≡

```
  void decrease_key_dheap(v)
      Vertex *v;
  {
    register unsigned long p, f;

    f = v→posicao;
    p = teto(f − 1, D);      /* pai de um vértice é encontrado na posição ⌈(f − 1)/D⌉ */
    while ((p > 0) ∧ (Q(p)→dist > v→dist)) {
      Q(f) = Q(p);
      Q(f)→posicao = f;
```

```
    f = p;
    p = teto(f - 1, D);
  }
  Q(f) = v;
  Q(f)→posicao = f;
}
```

Lema 4.5 (operações D-heap): *Na implementação da fila de prioridade Q, utilizando um* D*-heap, cada operação* **insert** *e* **decrease-key** *gasta tempo $O(\log_D n)$ e a operação* **delete-min** *gasta tempo $O(D \log_D n)$, onde $D = \max\{2, \lceil m/n \rceil\}$.* ▮

Portanto, do teorema 3.5 e do lema 4.5, o tempo gasto pelo algoritmo de Dijkstra utilizando um D-heap, é

$$\underbrace{O(n \log_D n)}_{\text{insert}} + \underbrace{O(m \log_D n)}_{\text{decrease-key}} + \underbrace{O(nD \log_D n)}_{\text{delete-min}} = O(m \log_D n).$$

Note que o valor escolhido para D na implementação é ótimo, pois para grafos esparsos, isto é, quando $m = O(n)$, o tempo gasto é $O(n \log n)$, e para grafos não-esparsos, isto é, quando $m = \Omega(n^{1+\epsilon})$ para algum $\epsilon > 0$, o tempo gasto é $O(m \log_D n) = O((m \log n)/(\log D)) = O((m \log n)/(\log n^{\epsilon})) = O((m \log n)/(\epsilon \log n)) = O(m/\epsilon) = O(m)$. A última igualdade é verdadeira desde que ϵ seja uma constante. Portanto, o tempo gasto é $O(m)$ que é ótimo.

Teorema 4.6: *A implementação do algoritmo de Dijkstra que utiliza um* D*-heap resolve o problema do caminho mínimo em um grafo com n vértices e m arcos em tempo $O(m \log_D n)$.* ▮

4.3 Fibonacci heap

Um **fibonacci heap** é um conjunto de arborescências que respeitam as seguintes propriedades:

Propriedade 4.7 (ordem): *Para cada par (p, f), em uma mesma arborescência, onde p é o pai do vértice f, $d(p) \leq d(f)$ (análoga a propriedade 4.1).*

Propriedade 4.8 (descendentes): *Para cada vértice p, os filhos de p podem ser ordenados de maneira que o i-ésimo filho tenha pelo menos $i-2$ filhos.*

Dizemos que o **grau** de um vértice é o seu número de filhos. Como conseqüência da propriedade 4.8 pode-se notar que todo vértice de grau k possui pelo menos F_{k+2} descendentes (incluindo ele mesmo), onde F_k é o número de fibonacci,

$$F_k = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ F_{k-1} + F_{k-2} & \text{if } k \geq 2 \end{cases}$$

ou

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i.$$

dando a entender a origem do nome fibonacci heap.

A implementação de um fibonacci heap é feita através de uma lista ligada Q, como ilustrada na figura 4.3.

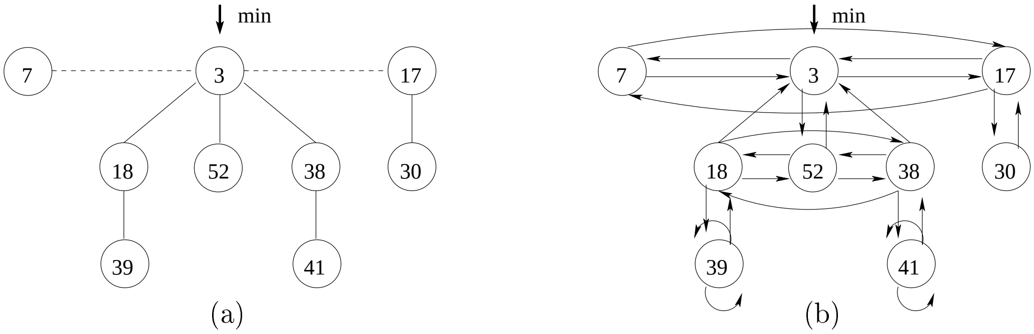


Figura 4.3: Em (a) é exibido um fibonacci heap composto por três arborescências. A linha pontilhada enfatiza o fato do fibonacci heap ser formado pelo conjunto das arborescências. Em (b) é ilustrada a representação em forma de lista, mostrando os ponteiros da estrutura.

A lista Q pode ser dividida em dois tipos de listas: (1) (*root*) uma lista formada pelas raízes das arborescências; e (2) (*child*) lista formada por filhos de um mesmo vértice. Cada vértice num fibonacci heap contém sete campos, que são: ponteiros *pai*, *filho*, *direita*, *esquerda*, que representam listas duplamente ligadas circulares (lista *root* e *child*); *grau_marca*, que guarda o número $2 * grau + marca$, onde o *grau* é o grau do vértice e *marca* tem valor 1 se o vértice (exceto o raiz) já perdeu algum filho, e 0 caso contrário. Os campos *predecessor* (ψ) e *dist* (d)

são os mesmos utilizados anteriormente. Note que a estrutura Vertex do SGB tem seis campos utilitários. Então, para implementar o fibonacci heap foi redefinida a estrutura **Vertex** do SGB para sete campos.

40 ⟨ Definições 12 ⟩ +≡

```
#define pai t.V
#define filho w.V
#define direita y.V
#define esquerda z.V
#define grau_marca x.I
```

As funções a seguir são responsáveis por inserir e remover vértices de uma lista. A função *insere_na_lista* insere na lista l o vértice v e a função *remove_da_lista* remove o vértice v da lista.

41 ⟨ Funções auxiliares 32 ⟩ +≡

```
void insere_na_lista(l, v)
      Vertex *l;
      Vertex *v;
{
   v->direita = l->direita;
   (l->direita)->esquerda = v;
   v->esquerda = l;
   l->direita = v;
}
void remove_da_lista(v)
      Vertex *v;
{
   (v->esquerda)->direita = v->direita;
   (v->direita)->esquerda = v->esquerda;
}
```

Para criar o fibonacci heap Q é utilizada a função *create_fheap*.

42 ⟨ Filas de prioridade 11 ⟩ +≡

```
void create_fheap(g)
      Graph *g;
{
   register Vertex *v;
```

```
    Q = Λ;
    for (v = g→vertices; v < g→vertices + g→n; v++) {
      v→direita = v→esquerda = v;
      v→grau_marca = 0;
      v→pai = v→filho = Λ;
    }
  }
```

As implementações das operações insert, delete-min e decrease-key são representadas pelas funções *insert_fheap, delete_min_fheap* e *decrease_key_fheap*.

43 ⟨ Filas de prioridade 11 ⟩ +≡

```
  void insert_fheap(Vertex *v);
  Vertex *delete_min_fheap( );
  void decrease_key_fheap(Vertex *v);
```

Inserir um vértice no fibonacci heap significa colocá-lo na lista *root*. No pior caso, essa operação gasta tempo $O(1)$.

44 ⟨ Filas de prioridade 11 ⟩ +≡

```
  void insert_fheap(v)
      Vertex *v;
  {
    if (Q ≡ Λ) {
      Q = v;
    }
    else {
      insere_na_lista(Q, v);      /* insere v na lista root */
      if (v→dist < Q→dist) Q = v;
    }
  }
```

O vértice com o menor potencial no fibonacci heap se encontra na primeira posição, ou seja, Q aponta para o vértice com potencial mínimo. Remover esse vértice implica em adicionar seus filhos à lista *root*, e em seguida é necessário utilizar o bloco ⟨ Consolidate 46 ⟩, para atualizar o número de arborescências e encontrar o novo vértice com potencial mínimo. Essa operação gasta tempo amortizado $O(\log qsize)$.

45 ⟨ Filas de prioridade 11 ⟩ +≡

```
Vertex *delete_min_fheap()
{
  register Vertex *min, *f, *v, *x, *y, *aux;
  register Vertex **A;
  register long i, d, Dn;
  min = Q;
  if (min ≠ Λ) {
    f = min→filho;
    while (f ≠ Λ) {      /* se min tem filhos */      /* para cada filho v de min */
      v = f;
      f = f→direita;
      if (v ≡ f) f = Λ;
      remove_da_lista(v);      /* remove v da lista child */
      insere_na_lista(Q, v);      /* insere v na lista root */
      v→pai = Λ;
    }
    remove_da_lista(min);      /* remove min da lista root */
    if (min→direita ≡ min) Q = Λ;
    else {
      Q = Q→direita;
      ⟨ Consolidate 46 ⟩
    }
  }
  return min;
}
```

O bloco ⟨ Consolidate 46 ⟩ é responsável por reduzir o número de raízes de Q, ou seja, reduzir o número de arborescências. Ele consiste em executar repetidamente os seguintes passos, até que todo vértice da lista *root* tenha graus distintos:

(1) encontre duas raízes x e y na lista *root* com o mesmo grau. Supõe-se que x→*dist* $\leq$ y→*dist*; e

(2) junte (link) y a x, removendo y da lista *root*, e adicionando y como um filho de x.

A operação (2) é feita em ⟨ Link arborescência y com a arborescência x 47 ⟩

46 ⟨ Consolidate 46 ⟩ ≡

```
Dn = 1 + 8 * sizeof(long);      /* 1+log(n): número máximo de arborescências */
```

```
if ((A = (Vertex **) malloc(Dn * sizeof(Vertex *))) ≡ Λ) {
   printf("%s\n", err_message[ERROR_2]);
   exit(0);
}
for (i = 0; i < Dn; i++)      /* A[d] guarda vértices de grau d */
   A[i] = Λ;
v = Q;
do {
   x = v;
   v = v→direita;
   if (x ≡ v) v = Λ;
   remove_da_lista(x);      /* remove x da lista root */
   d = (int) x→grau_marca/2;      /* 2 * grau + marca */
   while ((d < Dn) ∧ (A[d] ≠ Λ)) {
      y = A[d];
      if (x→dist > y→dist) {
         aux = x;
         x = y;
         y = aux;
      }
      ⟨ Link arborescência y com a arborescência x 47 ⟩
      A[d] = Λ;
      d++;
   }
   A[d] = x→direita = x→esquerda = x;
} while (v ≠ Λ);
Q = Λ;
for (i = 0; i < Dn; i++) {
   if (A[i] ≠ Λ) {
      if (Q ≡ Λ) Q = A[i]→direita = A[i]→esquerda = A[i];
      else {
         insere_na_lista(Q, A[i]);      /* adiciona A[i] na lista de root */
         if (A[i]→dist < Q→dist) Q = A[i];
      }
   }
}
free(A);
```

Este código é citado nos blocos 45 e 46.

Este código é usado no bloco 45.

47 ⟨ Link arborescência y com a arborescência x 47 ⟩ ≡

```
remove_da_lista(y);      /* remove y da lista root */
f = x→filho;
if (f) {
  insere_na_lista(f, y);      /* insere y como filho de x */
}
else x→filho = y→esquerda = y→direita = y;
y→pai = x;
x→grau_marca += 2;
y→grau_marca = (y→grau_marca ≫ 1) ≪ 1;      /* parte inteira da divisão por 2 e
    depois multiplicada por 2 */
```

Este código é citado no bloco 46.

Este código é usado no bloco 46.

Diminuir o potencial de um vértice v, ao visitá-lo, pode tornar necessário reposicioná-lo em Q, de modo que as propriedades 4.7 e 4.8 continuem sendo respeitadas. Isso é feito chamando-se ⟨ Remove v da lista de filhos de p 49 ⟩, onde p é o pai de v. Caso p esteja perdendo um segundo filho, ainda é necessário chamar a função *cascading_cut* para garantir a propriedade 4.8. Essa operação gasta tempo amortizado $O(1)$.

48 ⟨ Filas de prioridade 11 ⟩ +≡

```
void decrease_key_fheap(v)
    Vertex *v;
{
  register Vertex *p;
  p = v→pai;
  if ((p ≠ Λ) ∧ (v→dist < p→dist)) {
    ⟨ Remove v da lista de filhos de p 49 ⟩
    cascading_cut(p);
  }
  if (v→dist < Q→dist) Q = v;
}
```

49 ⟨ Remove v da lista de filhos de p 49 ⟩ ≡

```
if (v→direita ≡ v) p→filho = Λ;
```

else {
  if (p→*filho* $\equiv v$) p→*filho* = (p→*filho*)→*direita*;
  remove_da_lista(v); /* remove v da lista de filhos de p */
} /* decrementa o grau de p */
p→*grau_marca* $-{=}$ 2;
insere_na_lista(Q, v); /* adiciona v na lista *root* */
v→*pai* = Λ;
v→*grau_marca* = (v→*grau_marca* $\gg 1) \ll 1$; /* parte inteira da divisão por 2 e depois multiplicada por 2 */

Este código é citado nos blocos 48 e 50.

Este código é usado nos blocos 48 e 50.

A função *cascading_cut* repete o processo de ⟨ Remove v da lista de filhos de p 49 ⟩ até encontrar um vértice que ainda não tenha perdido nenhum filho.

50 ⟨ Funções auxiliares 32 ⟩ +≡
void *cascading_cut*(v)
  Vertex $*v$;
{
  register Vertex $*p$;

  $p = v$→*pai*;
  if ($p \neq \Lambda$) {
    if (v→*grau_marca* % 2 $\equiv 0$) v→*grau_marca*++;
    else {
      ⟨ Remove v da lista de filhos de p 49 ⟩
      cascading_cut(p);
    }
  }
}

Lema 4.9 (operações fibonacci heap): *Na implementação da fila de prioridade Q, utilizando um fibonacci heap, as operações* **insert** *e* **decrease-key** *gastam tempo amortizado $O(1)$ e a operação* **delete-min** *gasta tempo amortizado $O(\log n)$.* ∎

Portanto, do teorema 3.5 e do lema 4.9, o tempo gasto pelo algoritmo de Dijkstra utilizando um fibonacci heap, é

$$\underbrace{O(n)}_{\text{insert}} + \underbrace{O(m)}_{\text{decrease-key}} + \underbrace{O(n \log n)}_{\text{delete-min}} = O(m + n \log n).$$

Teorema 4.10: *A implementação do algoritmo de Dijkstra que utiliza um fibonacci heap resolve o problema do caminho mínimo em um grafo com n vértices e m arcos em tempo $O(m + n \log n)$.* ▮

4.4 Bucket heap

Um **bucket heap** é uma estrutura de dados que mantêm uma seqüência de vértices num vetor Q de listas ligadas. Para cada posição k do vetor, $Q(k)$ é uma lista ligada de vértices. Em cada $Q(k)$, são mantidos os vértices com potencial igual a k[2].

Observe que nC, onde C é o comprimento do maior arco, é um limitante superior para qualquer distância no grafo. Portanto, são necessários no máximo $nC + 1$ buckets, já que as distâncias podem variar de 0 a nC. Porém, é possível diminuir o número de buckets para $C + 1$. Observe que: (1) $d(u) \leq d(v)$ para cada vértice u em S e v em Q (invariante (dk3)); e (2) para cada vértice w em Q, $d(w) = d(u) + c(u, w)$ para algum vértice u em S (invariantes (dk4) e (dk12)). Portanto, de (1) e (2), $d(w) = d(u) + c(u, w) \leq d(u) + C$, onde u é o vértice escolhido no caso 2 do algoritmo de Dijkstra. Em outras palavras, toda distância tentativa é limitada inferiormente por $d(u)$, e superiormente por $d(u) + C$. Conseqüentemente, $C + 1$ buckets são suficientes. Resultando na seguinte propriedade.

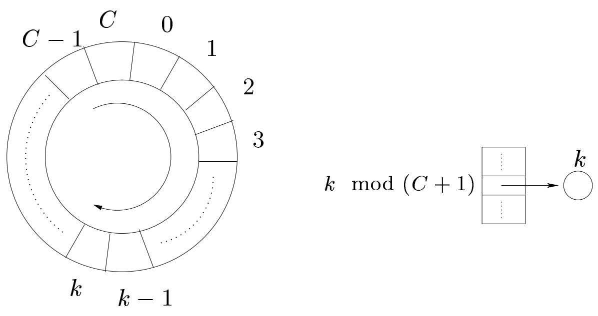
Propriedade 4.11 (ordem): *Cada bucket $Q(k \mod (C + 1))$ mantêm os vértices com potencial igual a k, onde C é o comprimento do maior arco.*

Os $C + 1$ buckets, numerados de $[0..C]$, simulam as $nC + 1$ posições funcionando de maneira circular. A figura 4.4 ilustra a organização dos buckets.

O ponteiro *pos_corrente* indica a posição do bucket que está sendo visitada.

52 ⟨Definições 12⟩ +≡
#**define** *pos_corrente* *w.I*

[2]Note que a ordem entre os vértices em $Q(k)$ não é importante



(a) (b)

Figura 4.4: A figura (a) enfatiza a organização circular dos buckets. (b) mostra que o bucket $k \mod (C+1)$ guarda o vértice com potencial k.

Para criar o bucket heap Q é utilizada a função *create_bkheap*. Redefinimos $Q(k)$ (a definição anterior está seção 4.1), de modo que seja o cabeça de lista do bucket k.

53 ⟨ Filas de prioridade 11 ⟩ +≡

```
#undef Q
#define Q(k)  (Q + k)
  void create_bkheap(g)
      Graph *g;
  {
    register unsigned long i;
    register Vertex *v;

    if ((Q = (Vertex *) malloc((C + 1) * sizeof(Vertex))) ≡ Λ) {
      printf("%s\n", err_message[ERROR_2]);
      exit(0);
    }
    for (i = 0; i < C + 1; i++) {
      Q(i)→direita = Q(i)→esquerda = Q(i);      /* endereço Q + i */
    }
    for (v = g→vertices; v < g→vertices + g→n; v++) {
      v→direita = v→esquerda = v;
    }
    Q→pos_corrente = 0;
  }
```

As implementações das operações insert, delete-min e decrease-key são representadas pelas funções *insert_bkheap*, *delete_min_bkheap* e *decrease_key_bkheap*.

54 ⟨ Filas de prioridade 11 ⟩ +≡

```
void insert_bkheap(Vertex *v);
Vertex *delete_min_bkheap();
void decrease_key_bkheap(Vertex *v);
```

Como a implementação é feita usando $C + 1$ buckets, cada vértice v é inserido na posição $v\to dist \mod (C+1)$. Essa operação gasta tempo $O(1)$.

55 ⟨ Filas de prioridade 11 ⟩ +≡

```
void insert_bkheap(v)
    Vertex *v;
{
  register unsigned long k;
  k = v->dist % (C + 1);
  insere_na_lista(Q(k), v);
}
```

O vértice com o menor potencial pode ser encontrado na primeira posição não-vazia de Q. No pior caso, essa operação gasta tempo $O(C)$.

56 ⟨ Filas de prioridade 11 ⟩ +≡

```
Vertex *delete_min_bkheap()
{
  register unsigned long k;
  Vertex *u;     /* procura o primeiro bucket não-vazio */
  for (k = Q->pos_corrente; Q(k) ≡ Q(k)->direita; k = ++k % (C + 1)) ;
  Q->pos_corrente = k;
  u = Q(k)->direita;
  remove_da_lista(u);
  return u;
}
```

Diminuir o potencial de um vértice, ao visitá-lo, pode tornar necessário reinseri-lo em Q, de modo que a propriedade 4.11 continue sendo respeitada. Essa operação gasta tempo $O(1)$.

57 ⟨Filas de prioridade 11⟩ +≡

```
void decrease_key_bkheap(v)
    Vertex *v;
{
  remove_da_lista(v);
  insert_bkheap(v);
}
```

Lema 4.12 (operações bucket heap): *Na implementação da fila de prioridade Q utilizando um bucket heap, as operações* insert *e* decrease-key *gastam tempo $O(1)$ e* delete-min *gasta tempo $O(C)$.* ∎

Portanto, do teorema 3.5 e do lema 4.12, o tempo gasto pelo algoritmo de Dijkstra utilizando um bucket heap, é

$$\underbrace{O(n)}_{\text{insert}} + \underbrace{O(m)}_{\text{decrease-key}} + \underbrace{O(nC)}_{\text{delete-min}} = O(m + nC).$$

Teorema 4.13: *A implementação do algoritmo de Dijkstra que utiliza um bucket heap resolve o problema do caminho mínimo em um grafo com n vértices e m arcos em tempo $O(m + nC)$, onde C é o maior comprimento de um arco.* ∎

4.5 Radix heap

Um **radix heap**, assim como um bucket heap (seção 4.4), é uma estrutura de dados que mantêm uma seqüência de vértices, num vetor Q, onde para cada posição k, $Q(k)$ é uma lista ligada de vértices. Porém, em vez de manter somente vértices com potencial k na $(k mod(C+1))$-ésima posição do bucket, são mantidos os vértices com potencial em um determinado intervalo. Na implementação do radix heap, os intervalos são consecutivos e têm largura $1, 1, 2, 4, 8, 16, \ldots$. Logo,

$$\begin{aligned} range(0) &= [0] \\ range(1) &= [1] \\ range(2) &= [2..3] \\ range(3) &= [4..7] \\ range(4) &= [8..15] \\ range(K) &= [2^{K-1}..2^K - 1] \end{aligned}$$

onde $K = \lceil \log(nC) \rceil$ e $range(k)$ é o intervalo do bucket $Q(k)$. Portanto, o número de buckets necessários é $1 + K$.

Propriedade 4.14 (ordem): *Se $range(k)$ é o intervalo do bucket $Q(k)$, então $Q(k)$ mantêm os vértices com potencial em $range(k)$.*

Os intervalos dos buckets mudam dinamicamente durante a execução do algoritmo, rearranjando-se de forma que os vértices com menor potencial fiquem nos buckets de largura 1. Por exemplo, supondo que o primeiro bucket não-vazio é o $Q(4)$, inicialmente, seu respectivo intervalo é [8..15], que é maior do que 1. Então, é necessário fazer a redistribuição dos intervalos da seguinte maneira.

$$\begin{aligned} range(0) &= [8] \\ range(1) &= [9] \\ range(2) &= [10..11] \\ range(3) &= [12..15] \\ range(4) &= \emptyset \end{aligned}$$

E também, é necessário redistribuir os vértices pertencentes a $Q(4)$ nos buckets $Q(0)$, $Q(1)$, $Q(2)$ e $Q(3)$, assim os vértices que podem ser examinados estão sempre em $Q(0)$ e $Q(1)$.

Na implementação, é suficiente guardar apenas o valor inicial do intervalo. Esse valor é mantido em *range* e cada vértice v será inserido no bucket $Q(bucket)$.

59 ⟨ Definições 12 ⟩ +≡

```
#define range  w.I
#define bucket  x.I
```

A função *log2* (x) calcula o $\lfloor \log_2 x \rfloor$ para $x \geq 1$.

60 ⟨ Funções auxiliares 32 ⟩ +≡

```
int log2(x)
    unsigned long x;
{
  register int lg = -1;
  do {
    x = x >> 1;
    lg++;
  } while (x);
  return lg;
}
```

Para criar o radix heap Q é utilizada a função *create_rxheap*.

61 ⟨Filas de prioridade 11⟩ +≡

```
void create_rxheap(g)
    Graph *g;
{
  register int i, K, range_len;
  register Vertex *v;
  K = log2(infinito) + 1;    /* número máximo de buckets */
  if ((Q = (Vertex *) malloc((1 + K) * sizeof(Vertex))) ≡ Λ) {
    printf("%s\n", err_message[ERROR_2]);
    exit(0);
  }
  Q(0)→direita = Q(0)→esquerda = Q(0);
  Q(0)→range = 0;
  for (i = range_len = 1; i ≤ K; i++) {
    Q(i)→direita = Q(i)→esquerda = Q(i);
    Q(i)→range = Q(i − 1)→range + range_len;
    if (i ≠ 1) range_len <<= 1;    /* range_len = range_len * 2 */
  }
  for (v = g→vertices; v < g→vertices + g→n; v++) {
    v→direita = v→esquerda = v;
    v→bucket = K;    /* indica que v está em Q(K) */
  }
}
```

As implementações das operações insert, delete-min e decrease-key são representadas pelas funções *insert_rxheap, delete_min_rxheap* e *decrease_key_rxheap.*

62 ⟨Filas de prioridade 11⟩ +≡

```
void insert_rxheap(Vertex *v);
Vertex *delete_min_rxheap();
void decrease_key_rxheap(Vertex *v);
```

A inserção dos vértices é feita respeitando-se os intervalos. No pior caso, essa operação gasta tempo $O(\log(nC))$.

63 ⟨Filas de prioridade 11⟩ +≡

```
void insert_rxheap(v)
```

```
    Vertex *v;
{
  register unsigned long k;
  for (k = v→bucket; k > 0; k−−) {      /* encontra o bucket para inserir v */
    if (v→dist ≥ Q(k)→range) break;
  }
  v→bucket = k;
  insere_na_lista(Q(k), v);
}
```

O vértice com o menor potencial é encontrado em $Q(0)$ ou $Q(1)$. Caso não haja vértices nessas posições é necessário fazer a redistribuição dos intervalos, e reinserir os vértices nos buckets corretos. No pior caso, essa operação gasta tempo $O(\log(nC))$.

64 ⟨ Filas de prioridade 11 ⟩ +≡

```
Vertex *delete_min_rxheap()
{
  register unsigned long range_len, k, i, min;
  register Vertex *u;
  if ((Q(0) ≡ Q(0)→direita) ∧ (Q(1) ≡ Q(1)→direita)) {
    ⟨ Encontra a posição k do primeiro bucket não-vazio 65 ⟩
    ⟨ Encontra o valor do menor potencial em Q(k) 66 ⟩
    ⟨ Redistribui os intervalos 67 ⟩
  }
  k = (Q(0) ≡ Q(0)→direita) ? 1 : 0;
  u = Q(k)→direita;
  remove_da_lista(u);
  return u;
}
```

65 ⟨ Encontra a posição k do primeiro bucket não-vazio 65 ⟩ ≡

```
for (k = 2; Q(k)→direita ≡ Q(k); k++) ;
```

Este código é usado no bloco 64.

66 ⟨ Encontra o valor do menor potencial em $Q(k)$ 66 ⟩ ≡

```
for (u = Q(k)→direita, min = infinito; u ≠ Q(k); u = u→direita)
  if (u→dist < min) min = u→dist;
```

Este código é usado no bloco 64.

67 ⟨Redistribui os intervalos 67⟩ ≡

```
Q(0)→range = min;
for (range_len = i = 1; i ≤ k − 1; i++) {
   Q(i)→range = Q(i − 1)→range + range_len;
   if (i ≠ 1) range_len ≪= 1;
}      /* o k-ésimo intervalo fica vazio */
Q(k)→range = infinito;
⟨Remove e distribui os vértices de Q(k) nos buckets anteriores 68⟩
```

Este código é usado no bloco 64.

68 ⟨Remove e distribui os vértices de $Q(k)$ nos buckets anteriores 68⟩ ≡

```
for (u = Q(k)→direita; Q(k) ≠ u; u = Q(k)→direita) {
   remove_da_lista(u);
   i = u→dist − Q(0)→range;
   u→bucket = (i ≡ 0) ? 0 : log2(i) + 1;
   insert_rxheap(u);
}
```

Este código é usado no bloco 67.

Diminuir o potencial de um vértice, ao visitá-lo, pode tornar necessário reinseri-lo em Q, de modo que a propriedade 4.14 continue sendo respeitada. Essa operação gasta tempo $O(1)$, pois o tempo gasto por *insert_rxheap*, já inclui as reinserções dos vértices.

69 ⟨Filas de prioridade 11⟩ +≡

```
void decrease_key_rxheap(v)
      Vertex *v;
{
   remove_da_lista(v);
   insert_rxheap(v);
}
```

Lema 4.15 (operações radix heap): *Na implementação da fila de prioridade Q, utilizando um radix heap, as operações* **insert** *e* **delete-min** *gastam tempo $O(\log(nC))$ e a operação* **decrease-key** *gasta tempo $O(1)$.* ∎

Portanto, do teorema 3.5 e do lema 4.15, o tempo gasto pelo algoritmo de Dijkstra utilizando um radix heap, é

$$\underbrace{O(n\log(nC))}_{\text{insert}} + \underbrace{O(m)}_{\text{decrease-key}} + \underbrace{O(n\log(nC))}_{\text{delete-min}} = O(m + n\log(nC)).$$

Teorema 4.16: *A implementação do algoritmo de Dijkstra que utiliza um radix heap resolve o problema do caminho mínimo em um grafo com n vértices e m arcos em tempo $O(m + n\log(nC))$, onde C é o maior comprimento de um arco.* ∎

4.6 Eficiência

A figura 4.5 resume o que foi visto neste capítulo, com relação aos tempos gasto para cada implementação de Q. No fibonacci heap, o tempo gasto é amortizado.

	heap	D-heap	fibonacci heap	bucket heap	radix heap
insert	$O(\log n)$	$O(\log_D n)$	$O(1)$	$O(1)$	$O(\log(nC))$
delete-min	$O(\log n)$	$O(D\log_D n)$	$O(\log n)$	$O(C)$	$O(\log(nC))$
decrease-key	$O(\log n)$	$O(\log_D n)$	$O(1)$	$O(1)$	$O(1)$
Dijkstra	$O(m\log n)$	$O(m\log_D n)$	$O(m + n\log n)$	$O(m + nC)$	$O(m + n\log(nC))$

Figura 4.5: Eficiência das implementações de uma fila de prioridade.