Relatório de Estudos

Shigueo Isotani

São Paulo, 8 de dezembro de 2000

Sumário

1. Estudo das diferentes implementações para o algoritmo de Dijkstra

2. Caminhos de Custo Mínimo

O problema do *Caminho de Custo Mínimo* aparece em diversos problemas práticos. Por exemplo, aparece na área de telecomunicações, quando queremos enviar uma mensagem de um ponto a outro, da maneira mais rápida possível; aparece também, na indústria de transportes, quando queremos enviar um veículo de um lugar para outro, da maneira mais barata possível.
Existem várias formulações do problema do *Caminho de Custo Mínimo*.
Neste relatório, estamos interessados na seguinte formulação:
Dado um grafo $G = (N, A)$, uma função custo c de A em $\mathcal{Z}_{\geq 0}$ e um vértice s, encontrar caminhos de custo mínimo de s a todo vértice $v \in N - s$.
O algoritmo mais conhecido para resolver esse problema é o *algoritmo de Dijkstra*.
Ele encontra caminhos de custo mínino de um vértice s a todos os outros vértices do grafo, quando os custos nos arcos são todos positivos.
A idéia básica desse algoritmo é visitar os vértices em ordem crescente de distância à partir de s. O algoritmo pode ser implementado utilizando-se várias estruturas de dados diferentes. Dependendo da estrutura de dados utilizada, pode-se melhorar a complexidade do algoritmo.

3. O objetivo do programa, aqui descrito, é fazer um estudo comparativo do tempo de execução do algoritmo de *Dijkstra* implementado com estruturas de dados diferentes.
O programa está escrito em *CWEB* - LaTeX, utilizando a estrutura do *StanfordGraphBase*.
O grafo **default** de teste é um grafo *orientado*, com 1000 vértices, 50000 arcos, com custo mínimo dos arcos igual a 1 e máximo igual a 10 e a "semente" para gerar um número randômico igual a 0. Esse grafo é gerado randomicamente pela subrotina *random_graph*, descrita no módulo `GB_RAND`. Os valores **default** podem ser mudados, passando parâmetros na linha de comando. Os parâmetros que podem ser usados são:

-n$< N >$ número de vértices do grafo.
-m$< N >$ número de arcos do grafo.
-cmin$< N >$ custo mímino de um arco.
-cmax$< N >$ custo máximo de um arco.
-s$< N >$ semente para um número randômico.
-o{0,1} 0 indica que o grafo é não-orientado,
1 indica que o grafo é orientado.

E se a opção -f$< NomeDoArquivo.gb >$ for utilizada, o programa utiliza como grafo teste o arquivo $< NomeDoArquivo.gb >$[1].

```
#include <time.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "gb_graph.h"
#include "gb_save.h"
#include "gb_rand.h"
#include "prototipos.h"
  ⟨ Variáveis Globais 4 ⟩
  ⟨ Funções 16 ⟩
  ⟨ Subrotinas que testam se o resultado está correto 18 ⟩
  ⟨ Fila de Prioridades 26 ⟩
  ⟨ Algoritmo 24 ⟩
  int main(int argc, char *argv[])
  {
    clock_t inicio, fim;
    unsigned long t;
    Vertex *s;    /* vértice inicial */
    char *grafo = Λ;    /* nome do arquivo que contém o grafo */
    unsigned long n = 1000;    /* número de vértices */
    unsigned long m = 50000;    /* número de arcos */
    unsigned long custo_min = 1;    /* custo mínimo de um arco */
    unsigned long custo_max = 10;    /* custo máximo de um arco */
    unsigned long semente = 0;    /* semente do número randômico */
    int o = 1;    /* 1 indica grafo orientado 0 indica grafo não-orientado */
```

[1] Note que o arquivo tem que estar no formato do sgb.

```
  while (--argc) {
    if (sscanf(argv[argc], "-n%ld", &n) ≡ 1) ;
    else if (sscanf(argv[argc], "-m%ld", &m) ≡ 1) ;
    else if (sscanf(argv[argc], "-cmin%ld", &custo_min) ≡ 1) ;
    else if (sscanf(argv[argc], "-cmax%ld", &custo_max) ≡ 1) ;
    else if (sscanf(argv[argc], "-s%ld", &semente) ≡ 1) ;
    else if (sscanf(argv[argc], "-o%d", &o) ≡ 1) ;
    else if (strncmp(argv[argc], "-f", 2) ≡ 0) grafo = argv[argc] + 2;
    else {
      printf("Uso:␣%s␣[-nN][-mN][-cminN][-cmaxN][-sN][-o{0,1}][-fgrafo]\n",
          argv[0]);
      return (0);
    }
  }
  ⟨ Define o grafo que será usado 5 ⟩;
  ⟨ Calcula o arco de maior custo e define o INFINITO 6 ⟩;
  ⟨ Pega o vértice inicial 7 ⟩;
  ⟨ Executa a versão com LDLC 9 ⟩;
  ⟨ Executa a versão com 2-Heap 10 ⟩;
  ⟨ Executa a versão com d-Heap 11 ⟩;
  ⟨ Executa a versão com Dial 12 ⟩;
  ⟨ Executa a versão com Radix Heap 13 ⟩;
  ⟨ Executa a versão com Fibonacci Heap 14 ⟩;
  gb_recycle(g);
  return (0);
}
```

4. ⟨ Variáveis Globais 4 ⟩ ≡

```
Graph *g;
unsigned long INFINITO;
void(*cria_fila_de_prioridade)( );
void(*insere_na_fila_de_prioridade)( );
Vertex *(*remove_min_fila_de_prioridade)( );
void(*atualiza_fila_de_prioridade)( );
int(*fila_de_prioridade_vazia)( );
void(*primeira_iteracao)( );
```

Veja tambm blocos 25, 32, 40, 48 e 59.

Este cdigo usado no bloco 3.

5. ⟨ Define o grafo que será usado 5 ⟩ ≡

```
if (grafo) {
  if ((g = restore_graph(grafo)) ≡ Λ) {
    printf("␣Erro␣na␣abertura␣do␣arquivo␣de␣entrada.\n");
    return (0);
```

```
    }
    m = g→m;
    n = g→n;
  }
  else {
    printf("Criando␣o␣grafo...\n");
    if ((g = random_graph(n, m, 0, 0, o, Λ, Λ, custo_min, custo_max, semente)) ≡ Λ) {
      printf("␣Erro␣na␣criaç~ao␣do␣grafo.\n");
      return (0);
    }
  }
```

Este cdigo usado no bloco 3.

6. ⟨ Calcula o arco de maior custo e define o INFINITO 6 ⟩ ≡

```
  C = arco_max();
  INFINITO = g→n * C + 1;
```

Este cdigo usado no bloco 3.

7. ⟨ Pega o vértice inicial 7 ⟩ ≡

```
  s = g→vertices;
```

Este cdigo usado no bloco 3.

8. Executa os algoritmos.

9. ⟨ Executa a versão com LDLC 9 ⟩ ≡

```
  cria_fila_de_prioridade = cria_lista;
  insere_na_fila_de_prioridade = insere_lista;
  remove_min_fila_de_prioridade = remove_min_lista;
  atualiza_fila_de_prioridade = atualiza_lista;
  fila_de_prioridade_vazia = lista_vazia;
  primeira_iteracao = primeira_iteracao_lista;
  printf("\nDijkstra␣LDLC\n");
  printf("Pressione␣ENTER␣para␣começar␣...\n");
  getchar();
  inicio = clock();
  Dijkstra(s);
  fim = clock();
  ⟨ Verifica a função potencial e a condição de otimalidade 15 ⟩;
  printf("|␣N␣|␣=␣%ld\t|␣A␣|␣=␣%ld\n", g→n, g→m);
  printf("Tempo:␣%.4f␣segundos\n", ((double) (fim − inicio))/CLOCKS_PER_SEC);
```

Este cdigo usado no bloco 3.

10. ⟨ Executa a versão com 2-Heap 10 ⟩ ≡

```
cria_fila_de_prioridade = cria_heap;
insere_na_fila_de_prioridade = insere_heap;
remove_min_fila_de_prioridade = remove_min_heap;
atualiza_fila_de_prioridade = atualiza_heap;
fila_de_prioridade_vazia = heap_vazio;
primeira_iteracao = primeira_iteracao_heap;
printf("\nDijkstra com 2_Heap\n");
printf("Pressione ENTER para começar ...\n");
getchar();
D = 2;
inicio = clock();
Dijkstra(s);
fim = clock();
⟨ Verifica a função potencial e a condição de otimalidade 15 ⟩;
printf("| N | = %ld\t| A | = %ld\tD = %ld\n", g->n, g->m, D);
printf("Tempo: %.4f segundos\n", ((double) (fim - inicio))/CLOCKS_PER_SEC);
```

Este cdigo usado no bloco 3.

11. ⟨ Executa a versão com d-Heap 11 ⟩ ≡

```
printf("\nDijkstra com d-Heap\n");
printf("Pressione ENTER para começar ...\n");
getchar();
t = teto(m, n);
D = t > 2 ? t : 2;
inicio = clock();
Dijkstra(s);
fim = clock();
⟨ Verifica a função potencial e a condição de otimalidade 15 ⟩;
printf("| N | = %ld\t| A | = %ld\tD = %ld\n", g->n, g->m, D);
printf("Tempo: %.4f segundos\n", ((double) (fim - inicio))/CLOCKS_PER_SEC);
```

Este cdigo usado no bloco 3.

12. ⟨ Executa a versão com Dial 12 ⟩ ≡

```
cria_fila_de_prioridade = cria_bucket;
insere_na_fila_de_prioridade = insere_bucket;
remove_min_fila_de_prioridade = remove_min_bucket;
atualiza_fila_de_prioridade = atualiza_bucket;
fila_de_prioridade_vazia = bucket_vazio;
primeira_iteracao = primeira_iteracao_bucket;
printf("\n\nDijkstra com buckets(Dial)\n");
printf("Pressione ENTER para começar ...\n");
getchar();
inicio = clock();
```

```
Dijkstra(s);
fim = clock();
⟨ Verifica a função potencial e a condição de otimalidade 15 ⟩;
printf("|␣N␣|␣=␣%ld\t|␣A␣|␣=␣%ld\t%ld␣Buckets\n", g->n, g->m, C + 1);
printf("Tempo:␣%.4f␣segundos\n", ((double) (fim − inicio))/CLOCKS_PER_SEC);
free(bucket);
```

Este cdigo usado no bloco 3.

13. ⟨ Executa a versão com Radix Heap 13 ⟩ ≡

```
cria_fila_de_prioridade = cria_radix_heap;
insere_na_fila_de_prioridade = insere_radix_heap;
remove_min_fila_de_prioridade = remove_min_radix_heap;
atualiza_fila_de_prioridade = atualiza_radix_heap;
fila_de_prioridade_vazia = radix_heap_vazio;
primeira_iteracao = primeira_iteracao_radix_heap;
printf("\n\nDijkstra␣com␣Radix␣Heap\n");
printf("Pressione␣ENTER␣para␣começar␣...\n");
getchar();
inicio = clock();
Dijkstra(s);
fim = clock();
⟨ Verifica a função potencial e a condição de otimalidade 15 ⟩;
printf("|␣N␣|␣=␣%ld\t|␣A␣|␣=␣%ld\t%d␣Buckets\n", g->n, g->m, K + 1);
printf("Tempo:␣%.4f␣segundos\n", ((double) (fim − inicio))/CLOCKS_PER_SEC);
for (t = 0; t ≤ K; t++) free(*(range + t));
free(range);
free(bucket);
```

Este cdigo usado no bloco 3.

14. ⟨ Executa a versão com Fibonacci Heap 14 ⟩ ≡

```
cria_fila_de_prioridade = cria_fibonacci_heap;
insere_na_fila_de_prioridade = insere_fibonacci_heap;
remove_min_fila_de_prioridade = remove_min_fibonacci_heap;
atualiza_fila_de_prioridade = atualiza_fibonacci_heap;
fila_de_prioridade_vazia = fibonacci_heap_vazio;
primeira_iteracao = primeira_iteracao_fibonacci_heap;
printf("\nDijkstra␣com␣Fibonacci␣Heap\n");
printf("Pressione␣ENTER␣para␣começar␣...\n");
getchar();
inicio = clock();
Dijkstra(s);
fim = clock();
⟨ Verifica a função potencial e a condição de otimalidade 15 ⟩;
printf("|␣N␣|␣=␣%ld\t|␣A␣|␣=␣%ld\n", g->n, g->m);
```

```
printf("Tempo:␣%.4f␣segundos\n", ((double) (fim − inicio))/CLOCKS_PER_SEC);
```

Este cdigo usado no bloco 3.

15. ⟨ Verifica a função potencial e a condição de otimalidade 15 ⟩ ≡

```
if (funcao_potencial_OK() ∧ verifica_caminhos_OK()) {
  printf(">>>>>>>>>>␣␣Funç˜ao␣potencial␣Ok␣␣<<<<<<<<<<\n");
  printf(">>>>>>>>>>␣␣Caminhos␣de␣custo␣mínimo␣Ok␣␣<<<<<<<<<<\n");
}
```

Este cdigo usado nos blocos 9, 10, 11, 12, 13 e 14.

16. A função *teto* recebe dois **unsigned long**, i e j; e devolve teto i dividido j .

⟨ Funções 16 ⟩ ≡

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

Veja tambm bloco 17.

Este cdigo usado no bloco 3.

17. A função *arco_max* devolve o custo do maior arco do grafo.

⟨ Funções 16 ⟩ +≡

```
unsigned long arco_max()
{
  register Vertex *v;
  register Arc *a;
  register unsigned long max = 0;
  for (v = g→vertices; v < g→vertices + g→n; v++)
    for (a = v→arcs; a; a = a→next)
      if (max < a→len) max = a→len;
  return max;
}
```

18. A função *funcao_potencial_OK* verifica se todos os vértices estão com um potencial válido. Se todos os potenciais forem válidos, a função devolve TRUE. Caso contrário, devolve FALSE e imprime um arco que não respeita a função potencial.

```
#define FALSE 0
#define TRUE 1
```

⟨ Subrotinas que testam se o resultado está correto 18 ⟩ ≡

```
int funcao_potencial_OK()
{
  register Vertex *v;
  register Vertex *u;
  register Arc *a;
```

```
  for (u = g->vertices; u < g->vertices + g->n; u++) {
    for (a = u->arcs; a; a = a->next) {
      v = a->tip;
      if (v->dist - u->dist > a->len) {       /* a função não respeita c */
        printf("\n********␣␣A␣funç~ao␣potencial␣n~ao␣é␣válida␣␣*********\n");
        printf("******␣Arco␣que␣n~ao␣respeita␣a␣funç~ao␣potencial␣******\n");
        printf("*******␣␣%s␣->␣%s␣␣----␣␣%ld␣-␣%ld␣>␣%ld␣␣************\n",
            u->name, v->name, v->dist, u->dist, a->len);
        return FALSE;
      }
    }
  }
  return TRUE;
}
```

Veja tambm bloco 19.

Este cdigo usado no bloco 3.

19. A subrotina *calcula_caminho* calcula o custo do caminho;
e a subrotina *verifica_caminhos_OK* verifica se o custo do caminho respeita a condição de otimalidade.

⟨ Subrotinas que testam se o resultado está correto 18 ⟩ +≡

```
unsigned long calcula_caminho(Vertex *v)
{
  register Vertex *pred;
  register Arc *a;
  unsigned long custo;
  if (v->predecessor ≡ Λ) {
    return 0;
  }     /* encontra a arco com ponta final em v */
  pred = v->predecessor;
  a = pred->arcs;
  while (a->tip ≠ v) a = a->next;
  custo = calcula_caminho(v->predecessor);
  return (custo + a->len);
}
int verifica_caminhos_OK()
{
  register Vertex *v;
  register unsigned long custo;
  for (v = g->vertices; v < g->vertices + g->n; v++)
    if ((v->dist ≠ INFINITO) ∧ ((custo = calcula_caminho(v)) ≠ v->dist)) {
      printf("\n******␣Custo␣do␣caminho␣␣de␣s␣a␣%s␣está␣errado␣******\n",
          v->name);
      printf("*********␣␣Errado␣:␣%ld\tCerto␣:␣%ld␣␣****************\n",
          custo, v->dist);
```

```
        return FALSE;
      }
  return TRUE;
}
```

20. Algoritmo de Dijkstra

21. Preliminares.
Em todo o programa vamos assumir que $\mid N \mid = n$, $\mid A \mid = m$, $C = max\{c_{uv}\}$ para todo $uv \in A$ e $INFINITO = n * C + 1$.
Vamos definir $CC(P)$ como o custo do passeio P.
Então, o caminho P é de *custo mínimo* se $CC(P) \leq CC(P')$ para todo P' com a mesma origem e mesmo destino.
Dado um grafo $G = (N, A)$,

uma *função potencial*, é uma função y de N em $\mathcal{Z}_{\geq 0}$.
uma *função custo*, é uma função c de A em $\mathcal{Z}_{\geq 0}$.

A *função potencial* y, respeita c(ou é viável) se $y(v) - y(u) \leq c_{uv}$ para todo $uv \in A$.
Uma função *psi* de N em N é uma função predecessor se $\psi(v) = v$ ou se $\psi(v) = u$ então $uv \in A$.
Um vértice v possui *distância tentativa(finita)* se $v\to dist \leq n*C$ e v está na fila de prioridades $\mathcal{L}$.
Observação. No contexto do programa, distâncias e custos são equivalentes.

22. Condição de otimalidade.

Lema 1 *Se P é um caminho de s a $v \in N - s$ e y uma função potencial viável então $CC(P) \geq y(v) - y(s)$.*

E como corolário imediato do lema 1.

Corolário 1 *Se P é um caminho de s a $v \in N - s$ e y uma função potencial viável tal que $CC(P) = y(v) - y(s)$ então P é um caminho de custo mínimo.*

23. Detalhes de implementação. A função custo c aparece, na implementação, como $a\to len$, onde a é um arco de A, e o campo *len* é um **long**.
A função potencial y aparece como $v\to dist$, onde v é um vértice de N e *dist* é um **long**. Portanto, uma função potencial viável, em termos da implementação, é da forma

$$v\to dist - u\to dist \leq a\to len, \text{ para todo } a \in A$$

(u e v são vértices e a é um arco com ponta inicial em u e final em v).
A função predecessor ψ aparece como $v\to predecessor$.
Como comentado anteriormente, podemos implementar o algoritmo de *Dijkstra* utilizando estruturas de dados diferentes.
Implementamos *Dijkstra* com as seguintes estruturas:

Implementação	*Complexidade*
LDLC	$O(n^2)$
Bucket	$O(m + nC)$
2-Heap	$O(m \log n)$
d-Heap	$O(m \log_d n)$, onde $d = \lceil m/n \rceil$
Radix Heap	$O(m + n \log(nC))$
Fibonacci Heap	$O(m + n \log n)$

24. Subrotina Dijkstra.

> *Entrada:* Um grafo com custos positivos nos arcos e um vértice s.
> *Devolve:* Uma "função" potencial viável *dist* tal que, para todo $v \in N - s$, quando $v\to dist \; - \; s\to dist \geq INFINITO$ então não existe caminho de s a v, caso contrário, $CC(P) = v\to dist \; - \; s\to dist$, onde P é um caminho de s a v.

Note que P pode ser determinado pela função *predecessor*.

⟨ Algoritmo 24 ⟩ ≡

```
void Dijkstra(Vertex *s)
{
  register Vertex *v;
  register Vertex *u;
  register Arc *a;
  primeira_iteracao(s);
  while (¬fila_de_prioridade_vazia()) {
    u = remove_min_fila_de_prioridade();
    for (a = u→arcs; a; a = a→next) {
      v = a→tip;
      if (v→dist − u→dist > a→len) {      /* se a função potencial não é viável */
        if (v→dist ≡ INFINITO) {      /* v não está na fila */
          v→dist = u→dist + a→len;
          v→predecessor = u;
          insere_na_fila_de_prioridade(v);
        }
        else {      /* v já estava na fila */
          v→dist = u→dist + a→len;
          v→predecessor = u;
          atualiza_fila_de_prioridade(v);
        }
      }
    }
  }
}
```

Este cdigo usado no bloco 3.

25. Lista Duplamente Ligada Circular (LDLC)

A fila de prioridades é implementada através de uma *lista ligada duplamente circular* com cabeça de lista.
Cada vértice utiliza 4 campos: *predecessor*, "lembra" o caminho de s a qualquer outro vértice do grafo; *dist*, guarda o custo do caminho de s a cada vértice do grafo; *direita* e *esquerda* ponteiros para o vértice a direita e a esquerda, na LDLC.

#define *predecessor* $w.V$
#define *dist* $x.I$
#define *direita* $y.V$
#define *esquerda* $z.V$

⟨ Variáveis Globais 4 ⟩ +≡ /* Lista de vértices com distância tentativa finita */
  Vertex $L[1]$;

26. L faz o papel de cabeça de lista.

⟨ Fila de Prioridades 26 ⟩ ≡

```
void cria_lista()
{
  L→direita = L→esquerda = L;
  L→dist = 0;
}
```

Veja tambm blocos 27, 28, 29, 30, 31, 33, 34, 35, 36, 37, 38, 39, 41, 42, 43, 44, 45, 46, 47, 49, 50, 51, 52, 53, 54, 55, 56, 57, 60, 61, 62, 63, 64, 65, 66, 67, 68 e 69.
Este cdigo usado no bloco 3.

27. Estamos inserindo os vértices de maneira ordenada, começando do final da lista.

⟨ Fila de Prioridades 26 ⟩ +≡

```
void insere_lista(Vertex *v)
{
  register Vertex *ant;
  register unsigned long d;
  ant = L→esquerda;
  d = v→dist;      /* insere ordenadamente */
  while (d < ant→dist) ant = ant→esquerda;
  v→esquerda = ant;
  v→direita = ant→direita;
  (v→direita)→esquerda = v;
  ant→direita = v;
}
```

28. ⟨ Fila de Prioridades 26 ⟩ +≡

```
Vertex *remove_min_lista()
{
  register Vertex *prox;
  prox = L→direita;
  if (prox ≡ L) return Λ;
  L→direita = prox→direita;
  (prox→direita)→esquerda = L;
  return prox;
}
```

29. Quando o campo *dist* de um vértice v diminui, temos que reinserir v na lista.

⟨ Fila de Prioridades 26 ⟩ +≡

```
void atualiza_lista(Vertex *v)
{
  register Vertex *ant;
  register unsigned long d;
  d = v->dist;     /* remove v da lista */
  ant = v->esquerda;
  ant->direita = v->direita;
  (v->direita)->esquerda = ant;     /* insere ordenadamente */
  while (d < ant->dist) ant = ant->esquerda;
  v->esquerda = ant;
  v->direita = ant->direita;
  (v->direita)->esquerda = v;
  ant->direita = v;
}
```

30. ⟨ Fila de Prioridades 26 ⟩ +≡

```
int lista_vazia()
{
  return (L->direita == L);
}
```

31. ⟨ Fila de Prioridades 26 ⟩ +≡

```
void primeira_iteracao_lista(Vertex *s)
{
  register Vertex *v;
  cria_lista();
  for (v = g->vertices; v < g->vertices + g->n; v++) {
    v->dist = INFINITO;
    v->predecessor = Λ;
    v->direita = v->esquerda = v;
  }
  s->dist = 0;
  insere_lista(s);
}
```

32. d-Heap

Um *heap* é uma fila de prioridades, que pode ser vista como uma árvore.
Em um d-Heap [1] cada nó tem no máximo d filhos.
Note que a *árvore binária* [2] é um 2-Heap.
Vamos mencionar algumas propriedades importantes do d-Heap:

Propriedade 1 *O pai, do vértice na posição i, é encontrado na posição $\lceil (i-1)/d \rceil$.*

Propriedade 2 *Os filhos, do vértice na posicão i, são encontrados nas posições $id-d+2, \ldots, min\{n, id+1\}$.*

E a propriedade mais importante é a seguinte:

Propriedade 3 *Para cada vértice i, $i\to dist \le j\to dist$, para todo j que é filho de i.*

Segue daí, que $H(1)\to dist$ deverá estar sempre com valor mínimo.

```
#define posicao v.I   /* indica qual o posição do vértice no heap */
#define H(i) (Heap + i)→u.V
⟨ Variáveis Globais 4 ⟩ +≡
  Vertex *Heap;   /* heap */
  unsigned long nelementos;   /* número de elementos corrente na fila de prioridades */
  unsigned long D;   /* D é o número máximo de filhos de um vértice */
```

33. ⟨ Fila de Prioridades 26 ⟩ +≡

```
  void cria_heap( )   /* cria um heap vazio */
  {
    Heap = g→vertices;
    nelementos = 0;
  }
```

34. Inserimos o vértice no final do *Heap*. É claro, que nessa hora, $v\to dist$ pode não estar satisfazendo a propriedade 3, então chamamos a subrotina *atualiza_heap*, que é a responsável por reestaurar essa propriedade.

```
⟨ Fila de Prioridades 26 ⟩ +≡
  void insere_heap(Vertex *v)
  {
    nelementos++;
    v→posicao = nelementos;
    atualiza_heap(v);
  }
```

35. ⟨ Fila de Prioridades 26 ⟩ +≡

```
  Vertex *remove_min_heap( )
  {
    register unsigned long p;
    register unsigned long f;
    register Vertex *v;
    Vertex *vmin;
```

```
    if (nelementos ≡ 0) return Λ;
    vmin = H(1);
    v = H(nelementos −−);
    p = 1;
    f = menorFilho(p);
    while (f ≤ nelementos) {
      if (H(f)→dist ≥ v→dist) break;
      H(p) = H(f);
      H(p)→posicao = p;
      p = f;
      f = menorFilho(p);
    }
    H(p) = v;
    H(p)→posicao = p;
    return vmin;
  }
```

36. Restaura a propriedade 3.

⟨ Fila de Prioridades 26 ⟩ +≡

```
  void atualiza_heap(Vertex *v)
  {
    register unsigned long p;
    register unsigned long f;
    f = v→posicao;
    p = teto(f − 1, D);
    while ((p > 0) ∧ (H(p)→dist > v→dist)) {
      H(f) = H(p);
      H(f)→posicao = f;
      f = p;
      p = teto(f − 1, D);
    }
    H(f) = v;
    H(f)→posicao = f;
  }
```

37. ⟨ Fila de Prioridades 26 ⟩ +≡

```
  int heap_vazio()
  {    /* retorna zero se o heap estiver vazio e um número diferente de zero caso contrário */
    return (nelementos ≡ 0);
  }
```

38. Devolve o filho do vértice i que possui a menor distância tentativa.

⟨ Fila de Prioridades 26 ⟩ +≡

```
  unsigned long menorFilho(unsigned long i)
  {
    unsigned long j, min, primeiro_filho, ultimo_filho;
```

```
    primeiro_filho = i * D - D + 2;
    ultimo_filho = nelementos < i * D + 1 ? nelementos : i * D + 1;
    min = primeiro_filho;
    for (j = primeiro_filho + 1; j <= ultimo_filho; j++)
      if (H(min)->dist > H(j)->dist) min = j;
    return min;
  }
```

39. ⟨ Fila de Prioridades 26 ⟩ +≡

```
  void primeira_iteracao_heap(Vertex *s)
  {
    register Vertex *v;
    cria_heap();
    for (v = g->vertices; v < g->vertices + g->n; v++) {
      v->dist = INFINITO;
      v->predecessor = Λ;
      v->posicao = 0;
      v->u.V = Λ;
    }
    s->dist = 0;
    insere_heap(s);
  }
```

40. Dial

A idéia principal do algoritmo *Dial* [1] é manter os vértices com distâncias tentativas em *buckets*. Então, no $bucket(k)$ vão estar todos os vértices com distância tentativa igual a k.
Agora, suponha que o $bucket(k)$ é o primeiro *bucket* não vazio. Então, cada vértice nesse *bucket* tem a distância tentativa mínima, pois, as distâncias não decrescem. Assim, podemos visitar cada um dos vértices que pertencem a ele. Se tivermos que atualizar a distância tentativa de algum vértice v, de d_1 para d_2, movemos o vértice v do $bucket(d_1)$, para o $bucket(d_2)$. E da próxima vez que precisarmos escolher um vértice com menor distância tentativa, nós começamos a visitar os *buckets* começando do $k+1$.
Note que nC é um limitante superior para qualquer distância no grafo, portanto, necessitamos de no máximo $nC+1$ *buckets*, já que as distâncias podem variar de 0 a nC.
Mas, sabemos que no final de cada iteração do algoritmo de *Dijkstra* vale que:
$v \to dist - u \to dist \le c_{uv}$ com $uv \in A$ e u o vértice removido da fila de prioridade no início da iteração.
Então, como $c_{uv} \le C$, vale que $v \to dist - u \to dist \le C$.
Em outras palavras, toda distância tentativa está no intervalo $[u \to dist,\ u \to dist + \mathrm{C}]$. Conseqüentemente, $C+1$ buckets são suficientes.
Nossa implementação usa $C+1$ *buckets*; e mantém no $bucket(v \to dist \bmod (C+1))$ todos os vértices v com distância tentativa $v \to dist$.
Cada $bucket(k)$ é uma lista duplamente ligada circular.

#define $bucket(k)\quad (bucket + k)$
$\langle$ Variáveis Globais 4 $\rangle$ $+\equiv$
  Vertex $*bucket$;
  unsigned long C;

41. Cria Bucket.
$\langle$ Fila de Prioridades 26 $\rangle$ $+\equiv$
  void *cria_bucket*( )
  {
    register unsigned long i;
    $bucket = ($**Vertex** $*)\ malloc((C+1) *$ **sizeof**(**Vertex**));
    for $(i = 0;\ i < C + 1;\ i{+}{+})\ bucket(i) \to direita = bucket(i) \to esquerda = bucket(i)$;
    $nelementos = 0$;
  }

42. Insere no Bucket.
$\langle$ Fila de Prioridades 26 $\rangle$ $+\equiv$
  void *insere_bucket*(**Vertex** $*v$)
  {
    register Vertex $*prox$;
    register unsigned long k;
    $k = v \to dist \,\%\, (C+1)$;
    $prox = bucket(k) \to direita$;
    $v \to direita = prox$;
    $prox \to esquerda = v$;
    $bucket(k) \to direita = v$;
    $v \to esquerda = bucket(k)$;
    $nelementos{+}{+}$;
  }

43. Remove do Bucket.

$\langle$ Fila de Prioridades 26 $\rangle$ +≡

```
Vertex *remove_bucket(unsigned long k)
{
   register Vertex *prox;
   prox = bucket(k)→direita;
   if (bucket(k) ≡ prox) return Λ;
   bucket(k)→direita = prox→direita;
   (prox→direita)→esquerda = bucket(k);
   return prox;
}
```

44. $\langle$ Fila de Prioridades 26 $\rangle$ +≡

```
Vertex *remove_min_bucket()
{
   static unsigned long k = 0;
   Vertex *u;
   while (¬(u = remove_bucket(k % (C + 1)))) k++;
   nelementos−−;
   return u;
}
```

45. Atualiza.

$\langle$ Fila de Prioridades 26 $\rangle$ +≡

```
void atualiza_bucket(Vertex *v)
{
   register Vertex *ant;
   ant = v→esquerda;
   ant→direita = v→direita;
   (v→direita)→esquerda = ant;
   nelementos−−;
   insere_bucket(v);
}
```

46. Verifica se o bucket está vazio.

$\langle$ Fila de Prioridades 26 $\rangle$ +≡

```
int bucket_vazio()
{
   return (nelementos ≡ 0);
}
```

47. Inicialização.

$\langle$ Fila de Prioridades 26 $\rangle$ +≡

```
void primeira_iteracao_bucket(Vertex *s)
{
   register Vertex *v;
```

```
    cria_bucket( );
    for (v = g→vertices; v < g→vertices + g→n; v++) {
        v→dist = INFINITO;
        v→predecessor = Λ;
        v→direita = v→esquerda = v;
    }
    s→dist = 0;
    insere_bucket(s);
}
```

48. Radix Heap

A implementação com *Radix Heap* [1] utiliza a mesma idéia dos *buckets* mas, ao invés de guardar somente vértices com distância tentativa k no $k-$ésimo *bucket*, como no Dial, nós podemos guardar no *bucket* k, todos os vértices com distância tentativa dentro de um determinado intervalo.
Na implementação do *Radix Heap* os intervalos tem largura $1, 1, 2, 4, 8, 16, \ldots$. É fácil ver que o número de buckets necessários é $1 + K$, onde $K = \lceil \log(nC) \rceil$.
Vamos representar o intervalo do bucket k por $range(k)$. E vamos guardar um vértice v no bucket k se $v \rightarrow dist \in range(k)$. Inicialmente, os buckets tem os seguintes intervalos:

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

Os intervalos dos buckets são mudados dinamicamente e nós rearranjamos os vértices de modo que os vértices com a menor distância tentativa fiquem no bucket de largura 1.
Suponha que o primeiro bucket não vazio é o $bucket(4)$, então seu respectivo intervalo é maior que um, por exemplo $[8, 15]$. Então, fazemos a seguinte redistribuiçao dos intervalos:

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

E depois, redistribuimos os vértices do $bucket(4)$ para os buckets 1, 2 e 3, respeitando os intervalos.

```
#define range(i, j)  *(*(range + i) + j)    /* intervalo [i, j] */
#define intervalo  v.I
⟨ Variáveis Globais 4 ⟩ +≡
  unsigned long **range;    /* range dos buckets */
  int K;    /* número de buckets */
```

49. ⟨ Fila de Prioridades 26 ⟩ +≡

```
  void cria_radix_heap()
  {
    register int i;
    K = (int) (ceil(log(g→n * C)/log(2)));
    range = (unsigned long **) malloc((1 + K) * sizeof(unsigned long *));
    bucket = (Vertex *) malloc((1 + K) * sizeof(Vertex));
    bucket(0)→direita = bucket(0)→esquerda = bucket(0);
    *(range + 0) = (unsigned long *) malloc(2 * sizeof(unsigned long));
    range(0, 0) = range(0, 1) = 0;
    for (i = 1; i < K + 1; i++) {
      bucket(i)→direita = bucket(i)→esquerda = bucket(i);
      *(range + i) = (unsigned long *) malloc(2 * sizeof(unsigned long));
      range(i, 0) = (unsigned long) (1 ≪ (i − 1));    /* 2^(i−1) */
      range(i, 1) = (unsigned long) ((1 ≪ i) − 1);    /* 2^i − 1 */
    }
    nelementos = 0;
  }
```

50. ⟨ Fila de Prioridades 26 ⟩ +≡

```
void insere_radix_heap(Vertex *v)
{
  register Vertex *prox;
  register unsigned long k;
  register unsigned long pos;
  pos = v→posicao;
  while (¬pertence_range(v, pos −−)) ;
  k = v→posicao = pos + 1;
  prox = bucket(k)→direita;
  v→direita = prox;
  prox→esquerda = v;
  bucket(k)→direita = v;
  v→esquerda = bucket(k);
  nelementos ++;
}
```

51. ⟨ Fila de Prioridades 26 ⟩ +≡

```
Vertex *remove_radix_heap(unsigned long k)
{
  register Vertex *prox;
  prox = bucket(k)→direita;
  if (bucket(k) ≡ prox) return Λ;
  bucket(k)→direita = prox→direita;
  (prox→direita)→esquerda = bucket(k);
  return prox;
}
Vertex *remove_min_radix_heap()
{
  static unsigned long k = 0;
  register Vertex *u = Λ;
  do {    /* remove quando o bucket tem tamnaho 1 */
    while (((k ≡ 0) ∨ (k ≡ 1)) ∧ ¬(u = remove_bucket(k))) k++;
    if (k ≥ 2) {
      while (bucket_k_vazio(k)) k++;    /* encontra o primeiro bucket não vazio */
      redistribui_range(k);    /* redistribui os intervalos */
      k = 0;
    }
  } while (u ≡ Λ);
  nelementos −−;
  return u;
}
```

52. ⟨ Fila de Prioridades 26 ⟩ +≡

```
void atualiza_radix_heap(Vertex *v)
{
  register Vertex *ant;    /* remove do bucket antigo */
```

```
    ant = v→esquerda;
    ant→direita = v→direita;
    (v→direita)→esquerda = ant;
    nelementos−−;
    insere_radix_heap(v);
  }
```

53. ⟨ Fila de Prioridades 26 ⟩ +≡

```
  int bucket_k_vazio(unsigned long k)
  {
    return (bucket(k)→direita ≡ bucket(k));
  }
```

54. ⟨ Fila de Prioridades 26 ⟩ +≡

```
  int radix_heap_vazio()
  {
    return (nelementos ≡ 0);
  }
```

55. ⟨ Fila de Prioridades 26 ⟩ +≡

```
  void redistribui_range(unsigned long k)
  {
    register Vertex *v;
    register unsigned long fim;    /* valor final de um intervalo */
    register unsigned long inicio;    /* valor inicial de um intervalo */
    register unsigned long i;    /* i-ésimo bucket/intervalo */
    register unsigned long largura;    /* largura de um intervalo */
    fim = range(0,0) = range(0,1) = range(k,0);
    for (i = largura = 1; i ≤ k − 1; i++, largura ≪= 1) {    /* largura = largura*2 */
      inicio = fim + 1;
      fim = inicio + largura − 1;
      range(i,0) = inicio;
      range(i,1) = fim;
    }    /* o k-ésimo intervalo fica vazio */
    range(k,0) = range(k,1) = (unsigned long) −1;
    while ((v = remove_radix_heap(k))) {
      nelementos−−;
      insere_radix_heap(v);
    }
  }
```

56. ⟨ Fila de Prioridades 26 ⟩ +≡

```
  int pertence_range(Vertex *v, unsigned long k)
  {
    return ((range(k,0) ≤ v→dist) ∧ (range(k,1) ≥ v→dist));
  }
```

57. $\langle$ Fila de Prioridades 26 $\rangle$ +≡

```
void primeira_iteracao_radix_heap(Vertex *s)
{
   register Vertex *v;
   cria_radix_heap();
   for (v = g->vertices; v < g->vertices + g->n; v++) {
      v->dist = INFINITO;
      v->predecessor = Λ;
      v->direita = v->esquerda = v;
      v->posicao = K;      /* indica o bucket que está o vértice */
   }
   s->dist = 0;
   s->posicao = 0;
   insere_radix_heap(s);
}
```

58. Fibonacci Heap

Fibonacci Heap [2] é uma estrutura que mantém um conjunto de árvores, que mudam dinamicamente ao longo da computação.
Estas árvores, além de satisfazerem a propriedade 3 também satisfazem as seguintes propriedades:

Propriedade 4 *Cada vértice (não raiz) perde no máximo um filho.*

Propriedade 5 *Para todo vértice v de grau d, seja $\{g_1, \ldots, g_d\}$ os graus de cada filho de v em ordem crescente, ou seja, $g_1 \leq g_2 \leq \ldots \leq g_d$, então vale que, $g_j \geq j - 2$ para todo j.*

Como conseqüência da prorpiedade 5, nós podemos provar por indução que todo vértice v de grau d tem pelo menos F_{k+2} filhos(incluindo ele mesmo). Onde F é o número de *fibonacci*,

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

Note que o número de *fibonacci* pode ser escrito da forma

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

Propriedade 6 *Todo vértice na Fibonacci Heap tem grau no máximo $1 + 2\log n$.*

59. Cada vértice na *Fibonacci Heap* contém cinco ponteiros, que são: *pai*, pai do vértice (ou Λ se o vértice é raiz); *filho*, um dos filhos do vértice; *direita* e *esquerda* os irmãos direita e esquerda; *predecessor*, mesma função das implementações anteriores; e mais dois campos: *dist*, já utilizado nas implementações anteriores e *grau_marca*, guarda um número que é $2 * grau + marca$, onde grau é o número de filhos do vértice e marca é 1 se o vértice(não raiz) já perdeu algum filho e 0 caso contrário.
Os filhos de cada vértice e as raízes de cada árvore, estão duplamente ligados pelos ponteiros *direita* e *esquerda* em listas circulares; os vértices nessas listas podem aparecer em qualquer ordem, e o ponteiro *filho* pode apontar para qualquer filho.
Observação. Note que a estrutura **Vertex** do *Stanford GraphBase* tem seis campos ulilitários. Então, para implementar o *Fibonacci Heap* nós redefinimos a estrutura **Vertex** para ter sete campos.

```
#define pai t.V
#define filho u.V
#define grau_marca v.I
```

⟨ Variáveis Globais 4 ⟩ +≡

```
Vertex *F;
```

60. ⟨ Fila de Prioridades 26 ⟩ +≡

```
void cria_fibonacci_heap()
{
  F = Λ;
  nelementos = 0;    /* número de elementos no fibonacci heap */
}
```

61. Insere o vértice v na lista circular de raízes.

⟨ Fila de Prioridades 26 ⟩ +≡

```
void insere_fibonacci_heap(Vertex *v)
{
  if (F ≡ Λ) {
    F = v;
  }
  else {      /* insere na lista de root */
    v→direita = F→direita;
    (F→direita)→esquerda = v;
    v→esquerda = F;
    F→direita = v;
    if (v→dist < F→dist) F = v;
  }
  nelementos++;
}
```

62. Adiciona cada filho do vértice *min* na lista de root e remove min dessa lista. Depois, consolida a lista de root, linkando as raízes de mesmo grau até que cada raíz da lista tenha grau diferente.

⟨ Fila de Prioridades 26 ⟩ +≡

```
Vertex *remove_min_fibonacci_heap()
{
  register Vertex *f;
  register Vertex *min;
  register Vertex *v;
  register Vertex *w;
  min = F;
  if (min ≠ Λ) {
    f = min→filho;
    if (f ≠ Λ) {      /* se min tem filhos */
      w = f;
      do {      /* para cada filho v de min */
        v = w;
        w = w→direita;
        if (v ≡ w) w = Λ;      /* remove v da lista de filhos */
        (v→esquerda)→direita = v→direita;
        (v→direita)→esquerda = v→esquerda;      /* insere v na lista de root */
        v→direita = F→direita;
        (F→direita)→esquerda = v;
        v→esquerda = F;
        F→direita = v;
        v→pai = Λ;
      } while (w ≠ Λ);
    }      /* remove min da lista de root */
    (min→direita)→esquerda = min→esquerda;
    (min→esquerda)→direita = min→direita;
    if (min→direita ≡ min) F = Λ;
    else {
      F = F→direita;
      consolida();
```

```
      }
      nelementos--;
    }
    return min;
  }
```

63. ⟨ Fila de Prioridades 26 ⟩ +≡

```
  void atualiza_fibonacci_heap(Vertex *v)
  {
    register Vertex *p;
    p = v->pai;
    if ((p ≠ Λ) ∧ (v->dist < p->dist)) {
      cut(v, p);
      cascading_cut(p);
    }
    if (v->dist < F->dist) F = v;
  }
```

64. ⟨ Fila de Prioridades 26 ⟩ +≡

```
  int fibonacci_heap_vazio( )
  {
    return (nelementos ≡ 0);
  }
```

65. Nós usamos a subrotina *link_fibonacci_heap* para unir duas árvores cujas raízes u e v possuem o mesmo grau.

⟨ Fila de Prioridades 26 ⟩ +≡

```
  void link_fibonacci_heap(Vertex *u, Vertex *v)
  {
    register Vertex *f;    /* remove u da lista de root */
    (u->esquerda)->direita = u->direita;
    (u->direita)->esquerda = u->esquerda;    /* insere u como filho de v */
    f = v->filho;
    if (f) {
      u->direita = f->direita;
      (f->direita)->esquerda = u;
      u->esquerda = f;
      f->direita = u;
    }
    else v->filho = u->esquerda = u->direita = u;
    u->pai = v;    /* aumenta o grau de v */
    v->grau_marca += 2;    /* marca de u recebe 0 */
    u->grau_marca = (u->grau_marca ≫ 1) ≪ 1;
  }
```

66. Reduz o número de árvores na *Fibonacci Heap*. A subrotina *consolida* consiste em executar repetidamente os seguintes passos até que toda raiz tenha grau distinto.

1. Encontrar duas raízes x e y com o mesmo grau na lista de raízes;
2. "Linkar" y com x. Estamos assumindo sempre que $x \rightarrow dist \le y \rightarrow dist$. O grau de x é incrementado e o campo *marca*, do vértice y, recebe com 0.

⟨ Fila de Prioridades 26 ⟩ +≡

```
void consolida()
{
  int Dn;
  register int i;
  register int d;
  register Vertex *w;
  register Vertex *x;
  register Vertex *y;
  register Vertex *aux;
  register Vertex **A;
  Dn = 1 + 8 * sizeof(long);      /* 1+log(n) */
  A = (Vertex **) malloc(Dn * sizeof(Vertex *));
  for (i = 0; i < Dn; i++) A[i] = Λ;
  w = F;
  do {
    x = w;
    w = w→direita;      /* remove x da lista de root */
    if (x ≡ w) w = Λ;
    (x→esquerda)→direita = x→direita;
    (x→direita)→esquerda = x→esquerda;
    d = (int) x→grau_marca/2;
    while ((d < Dn) ∧ (A[d] ≠ Λ)) {
      y = A[d];
      if (x→dist > y→dist) {
        aux = x;
        x = y;
        y = aux;
      }
      link_fibonacci_heap(y, x);
      A[d] = Λ;
      d++;
    }
    A[d] = x→direita = x→esquerda = x;
  } while (w ≠ Λ);      /* monta novamente a lista de root */
  F = Λ;
  for (i = 0; i < Dn; i++) {
    if (A[i] ≠ Λ) {
      if (F ≡ Λ) F = A[i]→direita = A[i]→esquerda = A[i];
      else {      /* adiciona A[i] na lista de root */
        A[i]→direita = F→direita;
        (F→direita)→esquerda = A[i];
        A[i]→esquerda = F;
        F→direita = A[i];
        if (A[i]→dist < F→dist) F = A[i];
```

```
          }
        }
      }
      free(A);
    }
```

67. Remove v da lista de filhos.

⟨ Fila de Prioridades 26 ⟩ +≡

```
void cut(Vertex *v, Vertex *p)
{      /* remove v da lista de filhos de p */
  if (v->direita ≡ v) p->filho = Λ;
  else {
    if (p->filho ≡ v) p->filho = (p->filho)->direita;
    (v->esquerda)->direita = v->direita;
    (v->direita)->esquerda = v->esquerda;
  }      /* decrementa o grau de p */
  p->grau_marca −= 2;      /* adiciona v na lista de root */
  v->direita = F->direita;
  (F->direita)->esquerda = v;
  v->esquerda = F;
  F->direita = v;
  v->pai = Λ;      /* marca de v recebe 0 */
  v->grau_marca = (v->grau_marca ≫ 1) ≪ 1;
}
```

68. ⟨ Fila de Prioridades 26 ⟩ +≡

```
void cascading_cut(Vertex *p)
{
  register Vertex *pp;

  pp = p->pai;
  if (pp ≠ Λ) {
    if (p->grau_marca % 2 ≡ 0)      /* marca de p igual a zero */
      p->grau_marca++;      /* marca de p recebe 1 */
    else {
      cut(p, pp);
      cascading_cut(pp);
    }
  }
}
```

69. ⟨ Fila de Prioridades 26 ⟩ +≡

```
void primeira_iteracao_fibonacci_heap(Vertex *s)
{
  register Vertex *v;

  cria_fibonacci_heap();
  for (v = g->vertices; v < g->vertices + g->n; v++) {
    v->dist = INFINITO;
    v->predecessor = Λ;
```

```
        v→direita = v→esquerda = v;
        v→grau_marca = 0;
        v→pai = v→filho = Λ;
    }
    s→dist = 0;
    insere_fibonacci_heap(s);
}
```

70. Conclusões

71. A implementação que obteve o melhor resultado nos testes foi a com *Bucket*, apesar de ter complexidade *pseudopolinomial*. Mas, com uma diferença bem pequena em relação as implementações com *Radix Heap* e 2-Heap. A implementação com *Fibonacci Heap* que tem a melhor complexidade não teve um bom desempenho. A que obteve o pior resultado foi a que usa LDLC.
Os testes foram feitos em um Pentium MMX 200 MHZ, rodando sistema operacional Linux Debian 2.1.
Os tempos estão em segundos. Foram feitos 10 testes para cada valor n e m. Os resultados mostrados nas tabelas abaixo, são o valor médio desses testes.
Os resultados dos testes foram:

$n = 2048$ e $m = 2097152$

cmax	LDLC	2-Heap	d-Heap	Dial/Bucket	Radix Heap	Fibonacci Heap
10	1.48	1.30	1.64	1.25	1.30	1.28
1000	2.37	1.34	1.88	1.31	1.32	1.34

$n = 4096$ e $m = 1398101$

cmax	LDLC	2-Heap	d-Heap	Dial/Bucket	Radix Heap	Fibonacci Heap
10	1.54	0.89	1.17	0.88	0.88	0.92
1000	3.17	0.94	1.31	0.89	0.91	0.96

$n = 10000$ e $m = 1000000$

cmax	LDLC	2-Heap	d-Heap	Dial/Bucket	Radix Heap	Fibonacci Heap
10	7.11	0.80	0.98	0.74	0.76	0.83
1000	16.68	0.90	1.18	0.76	0.81	0.91

Não foram feitos testes para grafos maiores, pois o tempo para criar o grafo era muito alto. Os resultados mostram que uma estrutura muito complexa, como a *fibonnacci heap*, nem sempre obtém o melhor resultado. O que pode estar influenciando nesse resulado, é o quanto de memória é utilizada pela estrutura.

72. Estudo do artigo

73. Introdução.
O artigo que nós estudamos foi:
Undirect Single Source Shortest Paths With Positive Integer Weights in Linear Time escrito por Mikkel Thorup [3].
Desde 1959, todos os algoritmos para resolver o problema do caminho de custo mínimo foram baseados no algoritmo de Dijkstra, ou seja, sempre visitar os vértices em ordem crescente de distância à partir de s. Mas, qualquer implementação do algoritmo de Dijkstra, ordena os vértices de acordo com suas distâncias em relação a s. No entanto, nós não sabemos ordenar em tempo linear.
Nesse artigo o autor descreve um algoritmo determinístico para resolver o problema do *Caminho de Custo Mínimo* para grafos não-dirigidos e com custos positivos nas arestas em tempo e espaço linear no número de arestas $O(m)$.
O algoritmo é baseado em uma estrutura hierárquica de *buckets*, onde o *bucket* ajuda a identificar pares de vértices que podem ser visitados em qualquer ordem.

74. Preliminares.
Vamos definir $d(v)$ como sendo a menor distância de s a v no grafo, então, para cada vértice v nós temos a distância tentativa $D(v) \geq d(v)$.
Um conjunto $S \subseteq V$ é tal que, para todo $v \in S$ então $D(v) = d(v)$.
Vamos denotar $\lfloor x/2^i \rfloor$ como $x \gg i$ para enfatizar que podemos calcular simplesmente dando um *shift* de i bits para direita.

75. Evitando o "gargalo" da ordenação.
A idéia apresentada pelo autor, para evitar o "gargalo" da ordenação, é a utlização de uma estrutura de buckets.
O seguinte lema nos diz como podemos evitar o "gargalo" da ordenação:

Lema 2 *Suponha que o conjunto de vértices V está dividido em subconjuntos disjuntos $V_1, \ldots, V_k$, e que todas as arestas entre os subconjuntos tem custo pelo menos δ. Suponha que para algum i, $v \in V_i/S$ e que $D(v) = minD(V_i/S) \leq minD(V/S) + \delta$. Então, $d(v) = D(v)$.*

76. Hierarquia de componentes.
A hierarquia de componentes é definida da seguinte forma: G_i é um subgrafo de G, cujo as arestas tem peso $< 2^i$. Então, G_0 consiste dos vértices isolados. No nível i na hierarquia de componentes, nós temos os componentes (subgrafos conexos máximos) de G_i.

77. Do Lema 2 e utilizando a idéia de hierarquia de componentes, chegamos no seguinte algoritmo:
Algoritmo Genérico.

Recebe um grafo(não-orientado) $G = (V, E)$, uma função custo c de E em $\mathcal{Z}_{\geq 0}$, um vértice s e devolve caminhos de custo mínimo de s a todos os outros vértices de G.
Cada iteração temos:
Uma função potencial $y \geq 0$, componentes $V_1, \ldots, V_k$, uma parte $S \subseteq N$ e um inteiro α.

1º **iteração.**
$y(i) = c_{ij}$ se $ij \in E$; caso contrário $y(i) = \infty$
$y(s) = 0$, $S = \{s\}$,
$\alpha = 0$
$V_i = \{v_i\}$ para todo $v_i \in V - \{s\}$,
$k = n - 1$,
$i = 1$,
$\beta = min\{y(i)\}$
CASO 1. $i = k + 1$

Faça $\alpha' = \alpha + 1$
Subcaso 1A $\alpha' = \lceil \log C \rceil$

Devolva y e pare.

Subcaso 1B $\alpha' \leq \lfloor \log C \rfloor$

Sejam $V'_1, \ldots, V'_{k'}$ os componentes de $G - E_{\geq 2^{\alpha'}}$
Comece nova iteração com $\alpha', k', eV'_1, \ldots, V'_{k'}$ nos papéis de $\alpha, k, eV_1, \ldots, V_k$.

CASO 2. $i \leq k$

$d = min\{y(j) \mid j \in V_i/S\}$
se $d \leq \beta + 2^{\alpha}$ faça
Seja $D = \{j \mid y(j) = d\}$
Visite todo j em D. Sempre guardando $\beta = min\{\beta, y(j)\}$
$S = S \cup D$
Comece nova iteração.

Observação. Ainda não está claro se o algoritmo funciona.

78. Referências

Referências

[1] T. L. Magnant R. K. Ahuja e J. Orlin, *Network flows*, Practice Hall.

[2] C. E. Leiserson T. H. Cormen e R. L. Rivest, *Introduction to algorithms*, MacGraw-Hill Book Company, 1999.

[3] M. Thorup, Undirect single source shortest paths with positive integer weights in linear time, *J. ACM.* (1999).

Índice Remissivo

Lista de Refinamentos

⟨ Algoritmo 24 ⟩ Usado no bloco 3.
⟨ Calcula o arco de maior custo e define o INFINITO 6 ⟩ Usado no bloco 3.
⟨ Define o grafo que será usado 5 ⟩ Usado no bloco 3.
⟨ Executa a versão com 2-Heap 10 ⟩ Usado no bloco 3.
⟨ Executa a versão com Dial 12 ⟩ Usado no bloco 3.
⟨ Executa a versão com Fibonacci Heap 14 ⟩ Usado no bloco 3.
⟨ Executa a versão com LDLC 9 ⟩ Usado no bloco 3.
⟨ Executa a versão com Radix Heap 13 ⟩ Usado no bloco 3.
⟨ Executa a versão com d-Heap 11 ⟩ Usado no bloco 3.
⟨ Fila de Prioridades 26, 27, 28, 29, 30, 31, 33, 34, 35, 36, 37, 38, 39, 41, 42, 43, 44, 45, 46, 47, 49, 50, 51, 52, 53, 54, 55, 56, 57, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69 ⟩ Usado no bloco 3.
⟨ Funções 16, 17 ⟩ Usado no bloco 3.
⟨ Pega o vértice inicial 7 ⟩ Usado no bloco 3.
⟨ Subrotinas que testam se o resultado está correto 18, 19 ⟩ Usado no bloco 3.
⟨ Variáveis Globais 4, 25, 32, 40, 48, 59 ⟩ Usado no bloco 3.
⟨ Verifica a função potencial e a condição de otimalidade 15 ⟩ Usado nos blocos 9, 10, 11, 12, 13 e 14.