Capítulo 3

Algoritmo de Dijkstra

Neste capítulo será descrito o celebrado algoritmo de Dijkstra [11] que resolve o problema do caminho mínimo, apresentado na seção 2.1, ou seja:

> **Problema** PCM(V, A, c, s): Dado um grafo (V, A), uma função comprimento c de A em $\mathbb{Z}_{\geq}$ e um vértice s, encontrar um caminho de comprimento mínimo de s até t, para cada vértice t em V. PCM

A idéia geral do algoritmo de Dijkstra para resolver o problema é a seguinte. O algoritmo é iterativo. No início de cada iteração tem-se dois conjuntos disjuntos de vértices S e Q. O algoritmo conhece caminhos de s a cada vértice em $S \cup Q$ e para os vértices em S o algoritmo sabe que o caminho conhecido tem comprimento mínimo. Cada iteração consiste em remover um vértice apropriado de Q, incluí-lo em S e examiná-lo, acrescentando, eventualmente, novos vértices a Q.

A descrição abaixo segue de perto a feita por Feofiloff [13].

3.1 Descrição

Para cada vértice t, acessível a partir de um dado vértice s, o algoritmo de Dijkstra encontra um caminho de s a t que tem comprimento mínimo. Da condição de otimalidade (corolário 2.3) tem-se que, para provar que um caminho P de um vértice s a um vértice t tem comprimento mínimo, basta exibir uma função potencial viável d tal que $c(P) = d(t) - d(s)$. Por outro lado, a condição de inacessibilidade (corolário 2.2) diz que se d é uma função potencial viável com $d(t) - d(s) = nC + 1$, então t não é acessível a partir de s, onde n é o número de vértices do grafo e $C := \max\{c(u, v) : uv \in A\}$. A correção do algoritmo de Dijkstra fornecerá a recíproca dessas condições.

Algoritmo de Dijkstra. Recebe um grafo (V, A), uma função comprimento c de A em $\mathbb{Z}_{\geq}$ e um vértice s e devolve uma função predecessor ψ e uma função potencial d que respeita c tais que, para cada vértice t, se t é acessível a partir de s, então ψ determina um caminho de s a t que tem comprimento $d(t) - d(s)$, caso contrário, $d(t) - d(s) = nC + 1$, onde $C := \max\{c(u, v) : uv \in A\}$ e $n := |V|$.

Cada iteração começa com uma função potencial d, uma função predecessor ψ, e partes S, Q e U de V.

No início da primeira iteração $d(s) = 0$ e $d(v) = nC + 1$ para cada vértice v distinto de s, $\psi(v) = v$ para cada vértice v, $S = \emptyset$, $Q = \{s\}$ e $U = V \setminus \{s\}$.

Cada iteração consiste no seguinte.

Caso 1: $Q = \emptyset$.

Devolva d e ψ e pare.

Caso 2: $Q \neq \emptyset$.

Escolha u em Q tal que $d(u)$ seja mínimo.

$S' := S \cup \{u\}$.

$Q' := Q \setminus \{u\}$.

$U' := U$.

Para cada v em V faça $d'(v) := d(v)$ e $\psi'(v) := \psi(v)$.

Para cada arco uv faça

Se $d'(v) > d(u) + c(u, v)$ então

$d'(v) := d(u) + c(u, v)$, $\psi'(v) := u$ e remova[1] v de U' e acrescente a Q'.

Comece nova iteração com d', ψ', S', Q' e U' nos papéis de d, ψ, S, Q e U. □

Nas interpretações intuitivas do algoritmo de Dijkstra é comum dizer que d guarda **distâncias tentativas**, que S é o conjunto dos vértices **examinados**, Q é o conjunto dos vértices **visitados** e que U é o conjunto dos vértices **adormecidos**. A figura 3.1 ilustra o algoritmo de Dijkstra em ação.

3.2 Invariantes

A correção do algoritmo de Dijkstra baseia-se nas demonstrações da validade de uma série de invariantes, enunciados a seguir. Estes invariantes são afirmações envolvendo os dados do

[1] É possível que v já pertença a Q' e não esteja em U'. Nesse caso, estas últimas duas instruções são redundantes.

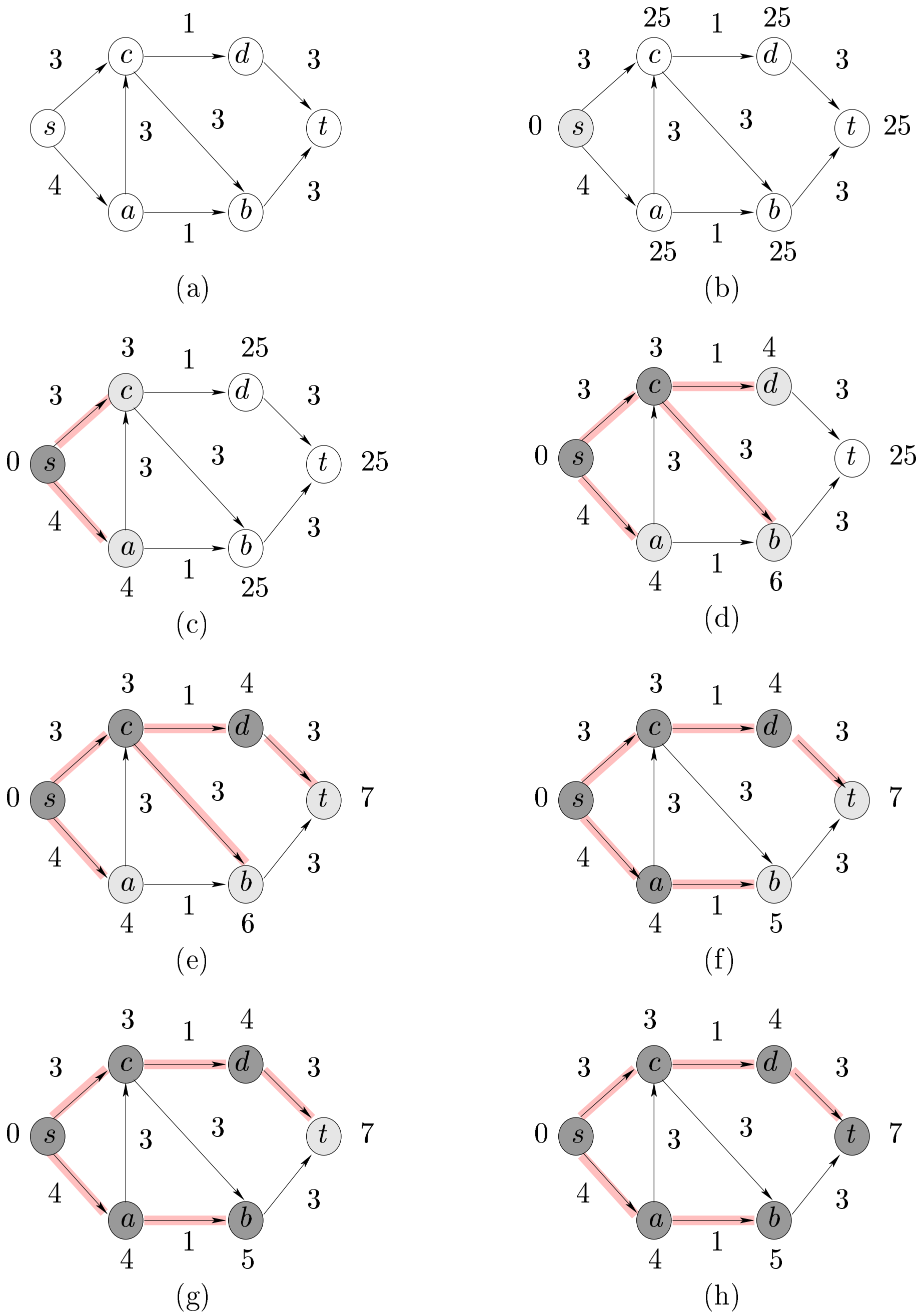

Figura 3.1: Execução do algoritmo de Dijkstra. O vértice inicial é s. (a) exibe um grafo com comprimento nos arcos. (b) mostra a situação no início da primeira iteração. Se um arco (u, v) está sombreado, então $\psi(v) = u$. Os potenciais são os números próximos a cada vértice. Os vértices pretos são os de S, os vértices cinzas são os de Q, e os vértices brancos estão em U. (c)-(g) exibem a situação após cada iteração do caso 2. Os valores finais da função potencial d, e da função predecessor ψ, são mostrados em (h).

problema V, A, c e s e os objetos d, ψ, S, Q e U. As afirmações são válidas no início de cada iteração do algoritmo e dizem como estes objetos se relacionam entre si e com os dados do problema.

Os invariantes estão agrupados conforme os objetos envolvidos.

Estrutura do grafo. Os dois invariantes a seguir envolvem somente as partes S, Q e U. A estrutura induzida por estas partições é ilustrada na figura 3.2(a).

(dk1) (partição) As partes S, Q e U formam uma partição de V.

(dk2) ($A(S, U) = \emptyset$) não existe arco uv com u em S e v em U.

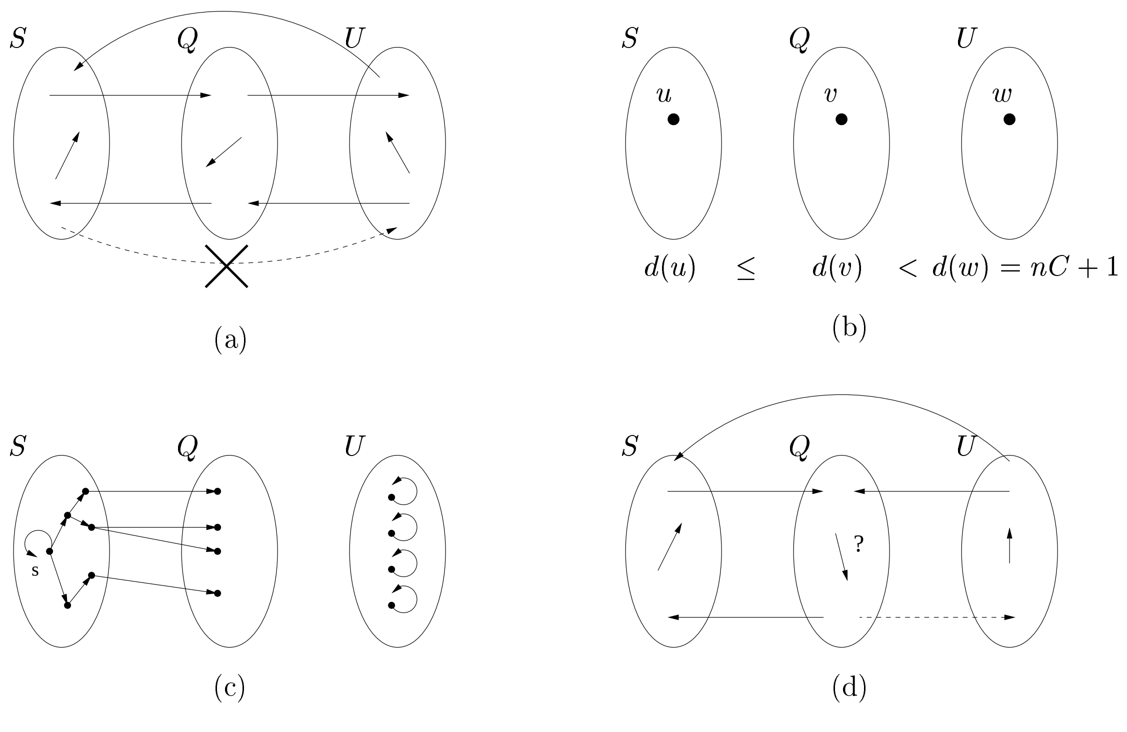

Figura 3.2: Ilustração dos invariantes. A figura (a) mostra a estrutura do grafo em relação à partição S, Q e U de V (invariantes (dk1) e (dk2)). (b) exibe a relação de ordem envolvendo os potenciais dos vértices e as partes S, Q e U (invariante (dk3)). A figura (c) mostra os arcos determinados pela função predecessor ψ (invariantes (dk4), (dk5) e (dk6)). Em (d) os arcos "pontilhados" são os que desrespeitam c, os com o símbolo "?" ao lado são os que podem ou não respeitar c, e o arcos "cheios" são os que respeitam c (invariantes de (dk7) a (dk11)).

Função potencial e $\{S, Q, U\}$. O próximo invariante reflete o fato do algoritmo examinar os vértices em ordem não-decrescente da distância a partir de s, ilustrado na figura 3.2(b).

(dk3) (monotonicidade) para cada u em S, v em Q e w em U vale que[2]

$$d(u) \leq d(v) < d(w) = nC + 1.$$

Função predecessor e $\{S, Q, U\}$. A estrutura da função predecessor em cada iteração, descrita pelos invariantes abaixo, está ilustrada na figura 3.2(c).

(dk4) ($\psi(Q) \subseteq S$) Para cada u em Q vale que $\psi(u)$ está em S.

(dk5) (identidade) Para cada u em U vale que $\psi(u) = u$.

(dk6) (estrutura arbórea de ψ) A função ψ restrita aos vértices em $S \cup Q$ determina uma arborescência em (V, A) com raiz s.

Arcos onde a função potencial respeita ou desrespeita c. Com excessão feita aos arcos com ambas as pontas em Q, o algoritmo sabe se d respeita ou 'desrespeita' c em cada um dos demais arcos do grafo. Isto não é nenhuma surpresa, tendo em vista que os vértices em Q tiveram seus potenciais alterados e os arcos com ambas as pontas em Q ainda não foram examinados. A situação encontra-se ilustrada na figura 3.2(d).

(dk7) (d respeita c em $A[S]$) Para cada arco uv com u e v em S vale que

$$d(v) - d(u) \leq c(u, v).$$

(dk8) (d respeita c em $A[U]$) Para cada arco uv com u e v em U vale que

$$d(v) - d(u) = (nC + 1) - (nC + 1) = 0 \leq c(u, v).$$

(dk9) (d respeita c em $A(S, Q)$ e $A(Q, S)$) Para cada arco uv com u em S e v em Q ou u em Q e v em S vale que

$$d(v) - d(u) \leq c(u, v).$$

(dk10) (d respeita c em $A(U, S)$ e $A(U, Q)$) Para cada arco uv com u em U e v em $S \cup Q$ vale que

$$d(v) - d(u) = d(v) - (nC + 1) < c(u, v).$$

(dk11) (d desrespeita c em $A(Q, U)$) Para cada arco uv com u em Q e v em U vale que

$$d(v) - d(u) = (nC + 1) - d(u) > c(u, v).$$

[2] A expressão "$d(u) \leq d(v) < d(w) = nC+1$" deve ser entendida como uma abreviação. Se algum dos conjuntos envolvidos é vazio, considere as desigualdades que fazem sentido.

Função potencial e função predecessor

(dk12) (folgas complementares) para cada arco uv tal que $\psi(v) = u$ vale que

$$d(v) - d(u) = c(u, v).$$

Para demonstrar que as afirmações acima são legítimos invariantes deve-se demonstrar que:

(a) as afirmações valem no início da primeira iteração; e

(b) se as afirmações valem no início de uma iteração em que ocorre o caso 2, então as afirmações também valem ao final da iteração com d', ψ', S', Q' e U' nos papéis de d, ψ, S, Q e U.

De (a) e (b) conclui-se que as afirmações também valem no início da última iteração; quando ocorre o caso 1. Supondo-se, é claro, que mais cedo ou mais tarde o caso 1 sempre ocorre.

É evidente que cada uma das afirmações valem no início da primeira iteração e não é difícil verificar (b) para cada umas das afirmações. A seguir estão as demonstrações de (dk1), (dk2), (dk3), (dk7) e (dk9) a título de ilustração. Nas demonstrações, são feitas referências ao procedimento de examinar um vértice ou arco como descrito na seção 2.4.

Demonstração de (dk1): Considere uma iteração em que ocorre o caso 2. No início da iteração, $\{S, Q, U\}$ é uma partição de V. Portanto, antes do algoritmo examinar o vértice u tem-se que $\{S', Q', U'\}$ é uma partição de V, já que $S' = S \setminus \{u\}$, $Q' = Q \cup \{u\}$, e $U' = U$. Durante o exame de cada arco, os vértices eventualmente removidos de U' são a seguir inseridos em Q'. Logo, no final da iteração, $\{S', Q', U'\}$ forma uma partição de V. ∎

Demonstração de (dk2): Considere uma iteração em que ocorre o caso 2. No início da iteração tem-se que $A(S, U) = \emptyset$. Imediatamente antes do vértice u ser examinado vale que $S' = S \cup \{u\}$ e $U' = U$. Do invariante (dk11) sabe-se que, para cada arco uv com u em S' e v em U',

$$d(v) - d(u) > c(u, v).$$

Assim, cada arco uv com v em U' será, durante o exame do arco uv, removido de U' e acrescentado a Q'. Portanto, no final da iteração, tem-se que $A(S', U') = \emptyset$. ∎

É importante notar que na demonstração de (dk3) e (dk9) que é utilizada a escolha apropriada do vértice u: $d(u) \leq d(v)$ para cada v em Q.

Demonstração de (dk3): Considere uma iteração em que ocorre o caso 2. Note que ao final da iteração tem-se que $d'(v) \neq d(v)$ se e somente se v está em $Q' \subseteq (Q \setminus \{u\}) \cup U$ e uv é um arco do grafo com $d(v) - d(u) > c(u, v)$. Ademais, se $d'(v) \neq d(v)$ então $d'(v) = d(u) + c(u, v) < d(v)$.

Do invariante (dk3) sabe-se que para cada x em S, y em Q e z em U vale que $d(x) \leq d(y) < d(z) = nC + 1$. Assim, pela escolha de u (e como c é uma função de A em $\mathbb{Z}_{\geq}$), após examinar o vértice u tem-se que

$$d'(x) \leq d(u) \leq d'(y) < d'(z) = nC + 1$$

para cada x em $S' = S \cup \{u\}$, y em Q' e z em U'. ∎

Demonstração de (dk7): No início da iteração tem-se que d respeita c em $A[S]$ (invariante (dk7)), em $A(S,Q)$, e em $A(Q,S)$. Desta forma, no final de uma iteração em que ocorre o caso 2 vale que d' respeita c em $A[S']$, pois $S' = S \cup \{u\}$ e $d'(v) = d(v)$ para todo v em S'. ∎

Demonstração de (dk9): Novamente, considere uma iteração em que ocorre o caso 2. Da demonstração do invariante (dk3), obtem-se que no final da iteração vale que $d'(x) - d'(y) \leq 0$ para cada x em $S' = S \cup \{u\}$, y em Q'. Portanto, como os comprimentos dos arcos são não-negativos, d' respeita c em $A(Q', S')$.

O processo de examinar u faz com que d' respeite c em cada arco com ponta inicial em u. Do invariante (dk9) sabe-se que d respeita c em $A(S,Q)$. Assim, como $d'(v) = d(v)$ para cada v em S' e $d'(v) \leq d(v)$ para cada v em Q', concluí-se que d' respeita c em $A(S', Q')$.

∎

3.3 Correção

A correção do algoritmo de Dijkstra é facilmente demonstrada através dos invariantes apresentados.

Teorema 3.1 (teorema da correção): *Para cada vértice t acessível a partir de s o algoritmo de Dijkstra devolve um caminho de s a t que tem comprimento mínimo.*

Demonstração: Suponha que ψ e d são as funções devolvidas pelo algoritmo. Quando o algoritmo pára tem-se que $Q = \emptyset$ e do invariante (dk1) vale que S e U é uma partição de V.

Ao final do algoritmo, S é o conjunto de vértices acessíveis a partir de s. De fato, pelo invariante (dk2) sabe-se que nenhum vértice de U é acessível a partir de s, já que $A(S,U) = \emptyset$, e pelo invariante (dk6) tem-se que para cada vértice t em S a função ψ determina um caminho de s a t, já que ψ determina uma arborescência em $(S, A[S])$ com raiz em s.

Como $Q = \emptyset$ e $A(S,U) = \emptyset$, então dos invariantes (dk7) a (dk10) conclui-se que a função potencial d é viável.

Resta apenas verificar que cada caminho P com início em s, e determinado por ψ, tem comprimento mínimo. Suponha que t em S é o término de P. Do invariante (dk12) obtem-se que $c(P) = d(t) - d(s)$. Finalmente, como d é viável, pela condição de otimalidade conclui-se que P tem comprimento mínimo. ∎

Um corolário importante da correção do algoritmo é a seguinte especialização do teorema da dualidade de programação linear ao PCM.

Teorema 3.2 (teorema da dualidade): *Seja (V, A) um grafo, c uma função comprimento de A em $\mathbb{Z}_{\geq}$ e s e t vértices do grafo. Se t é acessível a partir de s, então vale que*

$$\min\{c(P) : P \text{ é um caminho de } s \text{ a } t\} = \max\{d(t) - d(s) : d \text{ é uma função potencial viável}\}.$$

Demonstração: Do lema da dualidade 2.1 tem-se que para todo caminho P' e toda função potencial viável d' vale que $c(P') \geq d'(t) - d'(s)$. O algoritmo de Dijkstra devolve um caminho P e uma função potencial viável d tal que $c(P) = d(t) - d(s)$. Logo, P é um caminho que tem comprimento mínimo e d é uma função potencial viável para a qual a diferença de potencial entre s e t é máxima. ∎

3.4 Eficiência

Primeiro, é conveniente notar que d', ψ', S', Q' e U' foram introduzidos na descrição do algoritmo por meras razões técnicas: ajudam nas demonstrações de invariantes. Desta forma, em termos de eficiência, não há necessidade de levar em consideração as instruções que inicializam estes objetos, já que estas podem ser simplesmente eliminadas da descrição. A descrição pode ser feita inteiramente em termos de d, ψ, S, Q e U.

O número de ocorrências do caso 2 é no máximo n, pois em cada ocorrência é acrescentado um novo vértice a S e $\{S, Q, U\}$ é uma partição de V. Portanto, o número de iterações é no máximo $n + 1$.

As duas seguintes operações são as principais responsáveis pelo consumo de tempo assintótico do algoritmo:

Escolha de um vértice com potencial mínimo. Cada execução desta operação gasta tempo $O(n)$. Como o número de ocorrências do caso 2 é no máximo n, então o tempo total gasto pelo algoritmo para realizar essa operação é $O(n^2)$.

Atualização do potencial. Ao examinar um arco o algoritmo eventualmente diminui o potencial da ponta final. Essa atualização de potencial é realizada não mais que $|A(u)|$ vezes ao examinar o vértice u. Ao todo, o algoritmo pode realizar essa operação não mais que

$\sum_{u \in V} |A(u)| = m$ vezes. Desde que cada atualização seja feita em tempo constante, o algoritmo requer uma quantidade de tempo proporcional a m para atualizar potenciais.

Assim, o consumo de tempo do algoritmo no pior caso é $O(n^2 + m) = O(n^2)$. O teorema abaixo resume a discussão.

Teorema 3.3 (consumo de tempo): *O algoritmo de Dijkstra, quando executado, no modelo de comparação-adição, em um grafo com n vértices e m arcos, gasta tempo $O(n^2)$.* ∎

Para grafos densos, ou seja, grafos onde $m = \Omega(n^2)$, o consumo de tempo de $O(n^2)$ do algoritmo de Dijkstra é ótimo, pois, é necessário que todos os arcos do grafo sejam examinados. Entretanto, se $m = O(n^{2-\epsilon})$ para algum ϵ positivo, existem métodos sofisticados, como o heap de Johnson [24], o fibonacci heap de Fredman e Tarjan [16], que permitem diminuir o tempo gasto para encontrar um vértice com potencial mínimo, gerando assim implementações que consomem menos tempo para resolver o problema.

A maneira mais utilizada para realizar as operações acima é através de implementações de filas de prioridade, que será objeto de estudo do próximo capítulo.

3.5 Complexidade

Fredman e Tarjan [16] observaram que como o algoritmo de Dijkstra examina os vértices em ordem de distância a partir de s (invariante (dk3)) então o algoritmo está, implicitamente, ordenando estes valores (ver figura 3.3). Assim, qualquer implementação do algoritmo de Dijkstra para o modelo de comparação-adição realiza, no pior caso, $\Omega(n \log n)$ comparações. Portanto, qualquer implementação do algoritmo para o modelo de comparação-adição faz $\Omega(m + n \log n)$ operações elementares.

Teorema 3.4 (limitante inferior): *No modelo de comparação-adição, o algoritmo de Dijkstra, quando executado em um grafo com n vértices e m arcos, gasta tempo $\Omega(m + n \log n)$.* ∎

Por outro lado, Thorup [40] mostrou que um algoritmo linear para ordenação pode ser usado em uma implementação do algoritmo de Dijkstra que executa um número de operações proporcional ao tamanho do grafo.

No próximo capítulo será mostrada a implementação de Fredman e Tarjan [16] para modelo de comparação-adição que gasta tempo $O(m + n \log n)$. O algoritmo de Dijkstra com essa

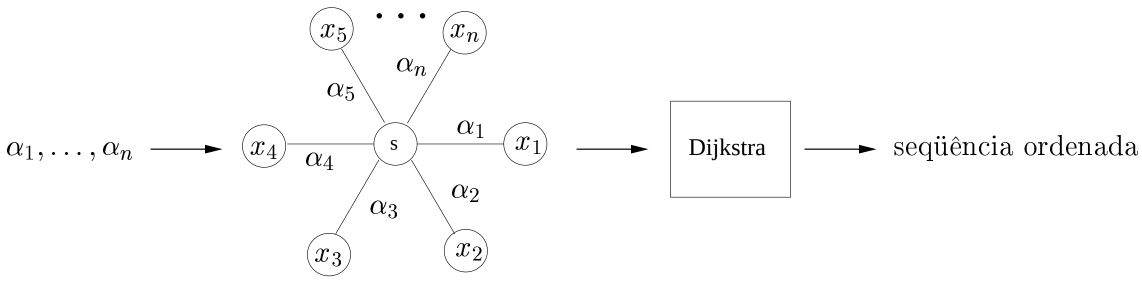

Figura 3.3: Esquema ilustrando como o algoritmo de Dijkstra pode ser utilizado para ordenar uma seqüência de números $\alpha_1, \ldots, \alpha_n$.

implementação é ótimo.

3.6 Versão em CWEB

Das partes S, Q e U descritas na seção 3.1, as partes S e U não entram na implementação. E a parte Q é representada através de uma fila de prioridade (seção 1.4).
Para criar Q é utilizada a operação `create_pq`, e as operações de Q serão representadas pelas funções `insert`, `delete_min` e `decrease_key`.

11 ⟨ Filas de prioridade 11 ⟩ ≡

```
void (*create_pq)( );      /* cria Q */
void (*insert)( );
Vertex *(*delete_min)( );
void (*decrease_key)( );
```

Veja também blocos 25, 26, 27, 28, 29, 33, 34, 35, 36, 38, 42, 43, 44, 45, 48, 53, 54, 55, 56, 57, 61, 62, 63, 64 e 69.

Este código é usado no bloco 9.

Os itens armazenados em Q são vértices, e como valor associado a cada um deles, temos o potencial. O potencial em cada vértice será representado pelo campo *dist*, e o predecessor pelo campo *pred*.

12 ⟨ Definições 12 ⟩ ≡

```
#define dist v.I
#define pred u.V
```

Veja também blocos 23, 40, 52, 59, 72, 89, 94, 104 e 119.

Este código é usado no bloco 9.

Será armazenado na variável *qsize*, o número de vértices em Q.

13 $\langle$ Variáveis globais 13 $\rangle \equiv$
unsigned long *qsize*;

Veja também blocos 24, 31, 71, 90, 103, 120 e 125.

Este código é usado no bloco 9.

14 $\langle$ Algoritmo de Dijkstra 14 $\rangle \equiv$

```
void dijkstra(g, s)
    Graph *g;
    Vertex *s;     /* vértice inicial */
{ ⟨Variáveis de Dijkstra 15⟩
  ⟨Inicializa d e ψ 16⟩
  ⟨Inicializa a fila Q com s 17⟩
  while (⟨Fila Q não está vazia 18⟩) {
    ⟨Escolha u em Q tal que d(u) seja mínimo 19⟩
    ⟨Examine o vértice u 20⟩
  }
}
```

Este código é citado no bloco 10.

Este código é usado no bloco 9.

15 $\langle$ Variáveis de Dijkstra 15 $\rangle \equiv$

```
register Vertex *v, *u;
register Arc *a;
```

Este código é usado no bloco 14.

No início da primeira iteração tem-se que $d(s) = 0$ e $d(v) = nC + 1$ para cada vértice v distinto de s, $\psi(v) = v$ para cada vértice v. Na implementação, $nC + 1$ será representado pela variável *infinito*.

16 $\langle$ Inicializa d e ψ 16 $\rangle \equiv$

```
for (v = g->vertices; v < g->vertices + g->n; v++) {
  v->dist = infinito;
  v->pred = v;
}
```

$s \to dist = 0;$
$num\_exam = 0;$
$atualiza\_fp = 0;$

Este código é usado no bloco 14.

Cria Q e a inicializa com o vértice inicial s.

17 ⟨ Inicializa a fila Q com s 17 ⟩ ≡
$create\_pq(g);$
$qsize = 0;$
$insert(s);$
$qsize{+}{+};$

Este código é usado no bloco 14.

18 ⟨ Fila Q não está vazia 18 ⟩ ≡
$(qsize > 0)$

Este código é usado no bloco 14.

19 ⟨ Escolha u em Q tal que $d(u)$ seja mínimo 19 ⟩ ≡
$u = delete\_min(\,);$
$qsize{-}{-};$

Este código é usado no bloco 14.

Examina todos os arcos da forma uv (seção 2.4).

20 ⟨ Examine o vértice u 20 ⟩ ≡
for $(a = u \to arcs;\ a;\ a = a \to next)$ {
  $v = a \to tip;$
  ⟨ Examine o arco uv 21 ⟩
}
$num\_exam{+}{+};$

Este código é usado no bloco 14.

Faz d respeitar c em uv (seção 2.4).

21 ⟨Examine o arco uv 21⟩ ≡

```
if (v->dist - u->dist > a->len) {      /* se a função potencial não é viável */
   atualiza_fp++;
   if (v->dist ≡ infinito) {      /* v não está na fila */
      v->dist = u->dist + a->len;
      v->pred = u;
      insert(v);
      qsize++;
   }
   else {      /* v já estava na fila */
      v->dist = u->dist + a->len;
      v->pred = u;
      decrease_key(v);
   }
}
```

Este código é usado no bloco 20.

O teorema a seguir resume o número de operações feitas pela implementação acima, para manipular a fila de prioridades Q.

> **Teorema 3.5** (número de operações): *O algoritmo de Dijkstra, quando executado em um grafo com n vértices e m arcos, realiza uma seqüência de n operações insert, n operações delete-min e no máximo m operações decrease-key.* ∎

Diferentes maneiras de se implementar as funções insert, delete_min e decrease_key serão estudadas no próximo capítulo.