# Основные операции для работы с d-кучами

Всюду далее, говоря о d-кучах, будем предполагать, что d является натуральным числом, не меньшим 2.

**Реализация основных операций для работы с d-кучами**

Рассмотрим d-кучу, реализованную на массиве key[1..n] ключей и массиве имен name[1..n] ее элементов. Функция minchild(i, key, n, d) позволяет для i-го узла n-элементной d-кучи с массивом ключей key находить его непосредственного потомка с минимальным ключом. Если у i-го узла нет потомков, то minchild(i, key, n, d) = i. Данная функция использует функции first\_child(n, d, i), last\_child(n, d, i), father(n, d, i), выдающие номера первого потомка, последнего потомка и родителя узла i n-элементной d-кучи соответственно. Значение father(n, d, 1) = 1, и если у i-го узла нет потомков, то first\_child(n, d, i) = 0, last\_child(n, d, i) = 0.

function minchild (i; var key; n, d);

begin

kf:= first\_child(n, d, i);

if kf = 0 then minchild:= i else

begin

kl:= last\_child(n, d, i);

min\_key:= key[kf];

minchild:= kf;

for j := kf +1 to kl do

if key[j]< min\_key then

begin

min\_key:= key[j];

minchild:= j;

end;

end;

end;

function first\_child(n, d, i);

begin

k:= (i-1)⋅d + 2;

if k>n then

first\_child:= 0

else

first\_child:= k;

end;

function last\_child(n,d,i);

begin

k:= first\_child(n,d,i);

if k= 0 then

last\_child:= 0

else

last\_child:= min{k+d-1,n};

end;

function father(n,d,i);

begin

father:= (i – 2)div d +1;

end;

procedure ПОГРУЖЕНИЕ( i; var name; var key; n,d);

begin

key0:= key[i];

name0:= name[i];

c:= minchild( i, key,n,d);

while (c ≠ i) & (key0 > key[c]) do

begin

key[i]:= key[c];

name[i]:= name[c];

{+}

i:= c; c:= minchild ( i, key, n,d);

end;

key[i]:= key0; name[i]:= name0;

{+}

end;

procedure ВСПЛЫТИЕ(i; var name; var key; n,d);

begin

key0:= key[i]; name0:= name[i]; p:= father(n,d,i);

while (i ≠ 1) and (key[p] > key0) do begin

key[i]:= key[p]; name[i]:= name[p];

{+}

i:= p; p:= father(n,d,i);

end;

key[i]:= key0; name[i]:= name0;

{+}

end;

procedure ИЗЪЯТИЕ\_МИНИМУМА (var name1; var key1;

var name; var key; var n; d);

begin

name1:= name[1]; key1:= key[1];

name[1]:= name[n]; key[1]:= key[n];

name[n]:= name1; key[n]:= key1;

n:= n-1;

if n>1 then ПОГРУЖЕНИЕ( 1, name,key,n,d);

end;

procedure ОБРАЗОВАTЬ\_ОЧЕРЕДЬ(var name; var key; n,d);

begin

for i:= n downto 1 do ПОГРУЖЕНИЕ (i);

end;

**Временная сложность операций с n–элементными d-кучами**

|  |  |
| --- | --- |
| ПОГРУЖЕНИЕ | O (d logdn) |
| ВСПЛЫТИЕ | O (logdn) |
| ИЗЪЯТИЕ\_МИНИМУМА | O (d logd n) |
| ОБРАЗОВАTЬ\_ОЧЕРЕДЬ | O (n) |

# Лабораторные работы

## 1. Нахождение кратчайших путей в графе

### Постановка задачи

Пусть G = (V, E, W) − ориентированный граф без петель со взвешенными ребрами, где множество вершин V={1, … , n}, множество ребер E ⊆ V×V, |E| = m, и весовая функция W(u,v) каждому ребру (u, v)∈E ставит в соответствие его вес − неотрицательное число. Требуется найти кратчайшие пути от заданной вершины s∈V до всех остальных вершин.

Если исходный граф не является ориентированным, то для использования описанных ниже алгоритмов следует превратить его в ориентированный, заменив каждое его ребро {u,v} на два ребра (u,v) и (v,u) того же веса.

Решением задачи будем считать два массива:

* массив dist[1..n], (dist[i] – кратчайшее расстояние от вершины s до вершины i).
* массив up[1..n], (up[i] – предпоследняя вершина в построенном кратчайшем пути из вершины s в вершину i).

В описываемых ниже алгоритмах “+∞” может быть заменено на любое число, превосходящее длину любого кратчайшего пути из вершины s в любую другую вершину графа G.

### Структура данных для представления графа

Множество OG(u)={v: (u,v)∈E} вершин графа G, являющихся концами ребер, выходящих из u, называется окрестностью вершины u. Окрестность вершины i представляется списком, элементами которого являются пары вида (j, W(i,j)). Сам граф будет представлен массивом ADJ[1..n] указателей на списки, соответствующие окрестностям вершин. Таким образом

* если |OG(i)|=0, то ADJ[i]=nil,
* в противном случае ADJ[i] является указателем на список, составленный из вершин окрестности OG(i); элемент этого списка представляет собой имя j очередной вершины j∈OG(i), вес w = W(i,j) ребра (i,j) и указатель next на следующую вершину из OG(i); если вся окрестность исчерпана, то next = nil.

Процедура FORM\_GRAPH заполнения данной структуры данных в соответствии с исходным графом G = (V, E, W) приведена ниже:

type

vtype = record

name: integer;

w: integer;

next: ^vtype;

end;

ADJtype = array[1..n] of ^vtype;

var

ADJ: ADJtype;

p: vtype;

procedure FORM\_GRAPH (var ADJ; G);

begin

for i:= 1 to n do begin

ADJ[i]:= nil;

for j∈OG(i) do begin

new(p);

p^.name:= j; p^.w:= W(i,j); p^.next:= ADJ[i];

ADJ[i]:= p;

end;

end;

end;

### Алгоритм Дейкстры, реализованный на основе d-кучи

Представим d-кучу массивом имен name[1..n] и массивом ключей key[1..n] так, что key[i] является текущей оценкой длины кратчайшего пути от вершины s к вершине name[i]. В данном алгоритме (см. [4]), представленном процедурой LDG\_DIJKSTRA\_D-HEAP, также используется массив index[1..n], который должен поддерживаться так, чтобы index[name[i]]:= i при i=1, …, n. Для достижения этой цели каждая строка “{+}” в псевдокоде операций ВСПЛЫТИЕ и ПОГРУЖЕНИЕ должна быть заменена строкой “index[name[i]]:= i;”

procedure LDG\_DIJKSTRA\_D-HEAP (var dist; var up; ADJ; n,d,s);

begin

for i:= 1 to n do begin

up[i]:= 0; dist[i]:= +∞; index[i]:= i; name[i]:= i; key[i]:= +∞;

end;

key[s]:= 0; nq:= n; ОБРАЗОВАTЬ\_ОЧЕРЕДЬ(name,key,nq,d);

while nq>0 do begin

ИЗЪЯТИЕ\_МИНИМУМА(name1,key1,name,key,nq,d);

i:= name1; dist[i]:= key1;

p:= ADJ[i].next;

while p ≠ nil do begin j:= p^.name; jq:= index[j];

{\*} if dist[jq] = +∞ then

if key[jq] > dist[i]+p^.w then begin

key[jq]:= dist[i]+p^.w;

ВСПЛЫТИЕ( jq,name,key,nq,d); up[j]:= i;

end;

p:= p^.next;

end;

end;

end;

Временная сложность алгоритма Дейкстры, реализованного на основе d-кучи, где d≥2, оценивается сверху величиной O((n+m)⋅log n), так как алгоритм производит n ИЗЪЯТИЙ\_МИНИМУМА и не более m ВСПЛЫТИЙ, каждое из которых осуществляется за время O(log n). Заметим также, что строку {\*} можно убрать без какого бы то ни было влияния на правильность работы алгоритма.

### Алгоритм Дейкстры, использующий метки

В рассматриваемой реализации данного алгоритма (см. [2]), представленной процедурой LDG\_DIJKSTRA\_MARK, массив h[1..n] является массивом меток: метка h[i]=0, если построение кратчайшего пути из вершины **s** в вершину **i** не завершено, и h[i]=1 в противном случае.

procedure LDG\_DIJKSTRA\_MARK (var dist; var up; ADJ; s);

begin

for i:= 1 to n do begin up[i]:= 0; dist[i]:= +∞; h[i]:= 0 end;

dist[s]:= 0; nq:= n;

while nq>0 do begin

c:= 1; while h[c] ≠ 0 do c:= c+1;

i:= c; for k:= c+1 to n do if h[k]= 0 then if dist[i]>dist[k] then i:= k;

h[i]:= 1; nq:= nq-1; p:= ADJ[i].next;

while p ≠ nil do begin

j:= p^.name;

{\*} if h[j] = 0 then

if dist[j] > dist[i]+p^.w then begin

dist[j] = dist[i]+p^.w; up[j]:= i;

end;

p:= p^.next;

end;

end;

end;

Временная сложность алгоритма Дейкстры, использующего метки, есть O(n2). Заметим, что также, как и раньше, строку {\*} можно убрать без какого бы то ни было влияния на правильность работы алгоритма.

### Алгоритм Форда–Беллмана

Алгоритм Форда–Беллмана (см. [5]), представленный процедурой LDG\_FORD\_BELLMAN, сначала полагает dist[s]=0 и dist[i]:= +∞ при всех i∈V\{s}, а затем (n-1) раз выполняет следующие действия:

для каждого ребра (i, j) ∈ E графа G = (V, E) такого, что j ≠ s, заменяет dist[j] на min{dist[j], dist[i]+W(i, j)}.

procedure LDG\_FORD\_BELLMAN (var dist; var up; ADJ; n,s);

begin

for i:= 1 to n do begin up[i]:= 0; dist[i]:= +∞ end; dist[s]:= 0;

for k:= 1 to n-1 do for i:= 1 to n do begin

p:= ADJ[i].next;

while p ≠ nil do begin

j:= p^.name;

if j ≠ s then if dist[j] > dist[i]+p^.w then begin

dist[j] := dist[i]+p^.w; up[j]:= i;

end;

p:= p^.next;

end;

end;

end;

Временная сложность алгоритма Форда-Беллмана есть O(n⋅m).

### Задания для лабораторной работы № 2

Предлагается попарное сравнение различных алгоритмов нахождения кратчайших путей от вершины s∈V до всех остальных вершин в графе G = (V, E), имеющем n вершин и m ребер.

Варианты выбора пары алгоритмов A и B для сравнения:

Вариант d=1, …, 10

А − алгоритм Дейкстры, реализованный на основе (d+1)–кучи,

В − алгоритм Дейкстры, реализованный на основе (d+2)–кучи;

Вариант d=11, …, 20

А − алгоритм Дейкстры, реализованный на основе (d-9)–кучи,

В − алгоритм Дейкстры, использующий метки;

Вариант d=21, …, 30

А − алгоритм Дейкстры, реализованный на основе (d-19)–кучи,

В − алгоритм Форда-Беллмана;

Вариант 31

А − алгоритм Дейкстры, использующий метки,

В − алгоритм Форда-Беллмана.

**Задание.**

1. Написать программу, реализующую алгоритм А и алгоритм В.
2. Написать программу, реализующую алгоритм А и алгоритм В, для проведения экспериментов, в которых можно выбирать:

* число n вершин и число m ребер графа,
* натуральные числа q и r, являющиеся соответственно нижней и верхней границей для весов ребер графа.

Выходом данной программы должно быть время работы ТА алгоритма А и время работы ТВ алгоритма В в секундах.

1. Провести эксперименты на основе следующих данных:
   1. n = 1, … ,104+1 с шагом 100, q = 1, r =106, количество ребер: а) m ≈ n2/10, б) m ≈ n2 (нарисовать графики функций TА(n) и ТВ(n) для обоих случаев);
   2. n = 101, … ,104+1 с шагом 100, q = 1, r  = 106, количество ребер: а) m ≈ 100⋅n, б) m ≈ 1000⋅n (нарисовать графики функций TА(n) и ТВ(n) для обоих случаев);
   3. n = 104+1, m = 0, … ,107 с шагом 105, q = 1, r = 106 (нарисовать графики функций TА(m) и ТВ(m) );
   4. n = 104+1, q = 1, r = 1, … ,200 с шагом 1, количество ребер: а) m ≈ n2, б) m ≈ 1000⋅n (нарисовать графики функций TА(r) и ТВ(r) для обоих случаев).
2. Сформулировать и обосновать вывод о том, в каких случаях целесообразно применять алгоритм А, а в каких − алгоритм В.

|  |  |  |
| --- | --- | --- |
| Вариант | Вариант пары алгоритмов | Вариант эксперимента |
| 1 | d=1 | 3.1 |
| 2 | d=2 | 3.2 |
| 3 | d=3 | 3.3 |
| 4 | d=4 | 3.4 |
| 5 | d=5 | 3.1 |
| 6 | 31 | 3.2 |
| 7 | d=11 | 3.3 |
| 8 | d=12 | 3.4 |
| 9 | d=13 | 3.1 |
| 10 | d=14 | 3.2 |
| 11 | d=15 | 3.3 |
| 12 | 31 | 3.4 |
| 13 | d=21 | 3.1 |
| 14 | d=22 | 3.2 |
| 15 | d=23 | 3.3 |
| 16 | d=24 | 3.4 |
| 17 | 31 | 3.1 |
| 18 | d=16 | 3.2 |
| 19 | d=26 | 3.3 |
| 20 | d=25 | 3.4 |

## Нахождение минимального остова графа

### Постановка задачи

Пусть G = (V, E, W) − неориентированный граф без петель со взвешенными ребрами и пусть множество вершин V={1, …, n}, множество ребер E ⊆ V×V, |E| = m и весовая функция W(u, v) каждому ребру (u, v)∈E ставит в соответствие неотрицательное число − его вес.

Требуется найти минимальный остов графа, то есть минимальное по весу поддерево графа G, содержащее все его вершины.

Решением задачи будем считать массив ET[1..n-1, 1..2], в котором пара (ET[i, 1], ET[i, 2]) является i-м ребром построенного минимального остовного дерева.

### Стратегии решения задачи

Рассмотрены несколько стратегий решения поставленной задачи, которые основываются на определенных правилах окрашивания вершин и ребер графа в два цвета (по традиции – синий и красный). В процессе окрашивания множество ребер, получивших к данному моменту синий цвет, представляет собой набор деревьев, являющихся фрагментами некоторого минимального остова. Ребра, получившие красный цвет, не входят ни в один минимальный остов. В итоге ребра, получившие синий цвет, представляют искомый остов.

***Стратегия* 1**(*Boruvka*)*.*

***Вначале*** *все вершины окрашиваются в синий цвет, а все ребра остаются неокрашенными.*

***Повторяющийся шаг***: *для каждого синего дерева выбирается инцидентное ему неокрашенное ребро минимальной стоимости, концы которого не являются вершинами одного и того же синего дерева, и окрашивается в синий цвет (если есть несколько ребер минимальной стоимости, то выбирается то, порядковый номер которого меньше).*

***Стратегия* 2**(*Kruskal*)*.*

***Вначале*** *все вершины окрашиваются в синий цвет, а все ребра остаются неокрашенными, множество ребер сортируется в порядке неубывания стоимостей.*

***Повторяющийся шаг*:** *выбирается очередное неокрашенное ребро в порядке неубывания стоимости*; *если оба его конца принадлежат одному и тому же синему дереву, то ребро окрашивается в красный цвет, в противном случае – в синий.*

***Стратегия* 3 *(Prim).***

***Вначале*** *в синий цвет окрашивается произвольная вершина, а все остальные вершины и ребра остаются неокрашенными.*

***Повторяющийся шаг:*** *выбирается инцидентное синему дереву неокрашенное ребро минимальной стоимости; если оба его конца принадлежат одному и тому же синему дереву, то ребро окрашивается в красный цвет, в противном случае – в синий.*

***Стратегия* 4 *(Yao).***

***Вначале*** *все вершины окрашиваются в синий цвет, а все ребра остаются неокрашенными.*

***Повторяющийся шаг*:** *выбирается произвольное синее дерево, являющееся максимальным по включению множеством синих ребер, образующих связный подграф, и инцидентное ему неокрашенное ребро минимальной стоимости; выбранное ребро красится в синий цвет.*

В алгоритмах, реализующих данные стратегии, будем применять разделенные множества с использованием рангов и сжатия путей (см. [5]). Для коллекции K разделенных множеств будем использовать операции:

* Найти(i, j, K) − найти имя i подмножества коллекции K, содержащее элемент j;
* Объединить(i, j) − объединить подмножества коллекции с именами i и j.

### Алгоритм Борувки

В данном алгоритме (см. [2]) используется представления исходного графа G массивом E[1..m] его ребер и массивом W[1..m] весов данных ребер.

procedure MSP\_BORUVKA(var ET; var mt; G; n,m);

begin

for s:= 1 to n do NME[s]:= 0;

Создать коллекцию K из n синглетонов множества {1, 2, … , n};

mt:= 0;

while FIND\_NME(NME,K,E,G,n,m) do for s:= 1 to n do if NME[s]>0 then begin

a=E[NME[s]][1]; b=E[NME[s]][2]; Найти(i, a, K); Найти(j, b, K);

if i ≠ j then begin mt:= mt+1; ET[mt]:= E[NME[s]]; Объединить(i,j,K); end;

NME[s]:= 0;

end;

end;

Функция FIND\_NME осуществляет повторяющийся шаг стратегии Борувки: для каждого синего дерева, представленного в коллекции множеством своих вершин, которое имеет имя s, находит инцидентное ему неокрашенное ребро минимальной стоимости с наименьшим номером NME(s), концы которого не являются вершинами одного итого же синего дерева. При этом функция FIND\_NME возвращает значение true, если хотя бы одно такое ребро найти удалось, и возвращает значение false, если этого сделать не получилось.

function FIND\_NME(var NME; K; E; G; n,m) : boolean;

begin

FIND\_NME = false;

for t:= 1 to m do begin

a=E[t][1]; b=E[t][2];

Найти(i,a,K); Найти(j,b,K);

if i≠j then begin

if NME[i]=0 then begin

NME[i]:= t; FIND\_NME=true;

end else if W[t]<W[NME[i]] then NME[i]:= t;

if NME[j]=0 then begin

NME[j]:= t; FIND\_NME=true;

end else if W[t]<W[NME[j]] then NME[j]:= t;

end;

end;

end;

Временная сложность алгоритма Борувки есть O(m⋅log n).

### Алгоритм Краскала

procedure MSP\_KRASKAL(var ET; var mt; G; n,m);

begin

Отсортировать массив E по невозрастанию весов его элементов;

Создать коллекцию из n синглетонов множества {1, 2, … , n};

mt:= 0;

for i:= 1 to m do begin

Найти имя a подмножества, содержащего элемент E[i][1];

Найти имя b подмножества, содержащего элемент E[i][2];

if a ≠ b then begin

Объединить подмножества коллекции с именами a и b;

mt:= mt+1; ET[mt]:= E[i];

end;

end;

end;

Временная сложность алгоритма Краскала есть O((m+n)⋅logn)).

### Алгоритм Прима

В данном алгоритме (см. [2]) используется представление исходного графа G окрестностями его вершин. Для окрестности i–ой вершины графа G = (V, E) мы будем использовать введенное обозначение OG(i). При этом заметим, что при непосредственном программировании следует использовать структуру ADJ[1..n]. В алгоритме используются также массивы

* a[1..n] (a[x] − вес минимального по весу ребра, соединяющего вершину x с построенным фрагментом минимального остова),
* b[1..n] (b[x] − имя второй вершины этого ребра) и
* VT[1..n] (VT[y]=1, если вершина y входит в построенный фрагмент минимального остова).

Алгоритм начинает свою работу с любой вершины u∈V. Для определенности положим, что u является первой вершиной.

procedure MSP\_PRIM(var ET; var mt; G; n,m);

begin

for i:= 1 to n do VT[i]:= 0; mt:= 0; u=1; VT[u]:= 1;

for x∈OG(u) do begin

a[x]:= W(x,u); b[x]:= u;

end else a[x]:= +∞;

while mt<n-1 do begin

u:= argmin{a[x] : VT[x]:= 0};

if a[u]:= +∞ then begin writeln(‘граф несвязен’); exit; end;

q:= b[u];

VT[u]:= 1; mt:= mt+1;

ET[mt][1]:= u; ET[mt][2]:= q;

for x∈OG(u) do if VT[x]=0 then if a[x]>W(x,u) then begin

a[x]:= W(x,u); b[x]:= u;

end;

end;

end;

Временная сложность алгоритма Прима есть O(n2).

**Примечание.** Используемый в псевдокоде знак +∞ обозначает число, которое больше веса любого ребра исходного графа G.

### Round Robin алгоритм

procedure MSP\_RRA(var ET; var mt; G; n,m);

begin

mt:= 0;

Создать пустую коллекцию K разделенных подмножеств множества V;

Создать пустой двусторонний список Q корневых элементов разделенных множеств из K;

for u∈V do begin Создать(u, K); Добавить u к концу списка Q;

Создать ленивую левостороннюю кучу H(u) из ребер, инцидентных u;

end;

while |Q|>1 do begin

Изъять первый элемент f из списка Q;

Найти элемент edge минимального веса в куче H(f);

a:= edge[1]; b:= edge[2]; Найти(i,a,K); Найти(j,b,K);

if i≠j then begin

mt:= mt+1; ET[mt]:= edge;

Удалить i и j из списка Q;

Объединить(i,j,K);

z:= корневой элемент подмножества, полученного объединением подмножеств с корневыми элементами и

i и j;

H(z):= ленивая левосторонняя куча, полученная слиянием куч H[i] и H[j];

Добавить z к концу списка Q;

end;

end;

end;

Временная сложность Round Robin алгоритма есть O(m⋅log log n).

**Примечание.** Элемент (ребро) в ленивой куче H[t] считается пустым, если концы этого ребра принадлежат одному и тому же множеству из коллекции K.

### Задания для лабораторной работы № 3

Предлагается попарное сравнение различных алгоритмов нахождения минимального по весу остовного дерева в графе G = (V, E), имеющего n вершин и m ребер.

Варианты выбора пары алгоритмов A и B для сравнения:

Вариант 1

* А − алгоритм Борувки,
* В − алгоритм Краскала;

Вариант 2

* А − алгоритм Борувки,
* В − алгоритм Прима;

Вариант 3

* А − алгоритм Прима,
* В − алгоритм Краскала;

Вариант 4

* А − алгоритм Борувки,
* В − Round Robin алгоритм.

**Задание.**

1. Написать программу, реализующую алгоритм А и алгоритм В.
2. Написать программу, реализующую алгоритм А и алгоритм В, для проведения экспериментов, в которых можно выбирать:

* число n вершин и число m ребер графа,
* натуральные числа q и r, являющиеся соответственно нижней и верхней границей для весов ребер графа.

Выходом данной программы должно быть время работы ТА алгоритма А и время работы ТВ алгоритма В в секундах.

1. Провести эксперименты на основе следующих данных:
   1. n = 1, … ,104+1 с шагом 100, q = 1, r =106, количество ребер: а) m ≈ n2/10, б) m ≈ n2 (нарисовать графики функций TА(n) и ТВ(n) для обоих случаев);
   2. n = 101, … ,104+1 с шагом 100, q = 1, r  = 106, количество ребер: а) m ≈ 100⋅n, б) m ≈ 1000⋅n (нарисовать графики функций TА(n) и ТВ(n) для обоих случаев);
   3. n = 104+1, m = 0, … ,107 с шагом 105, q = 1, r = 106 (нарисовать графики функций TА(m) и ТВ(m) );
   4. n = 104+1, q = 1, r = 1, … ,200 с шагом 1, количество ребер: а) m ≈ n2, б) m ≈ 1000⋅n (нарисовать графики функций TА(r) и ТВ(r) для обоих случаев).
2. Сформулировать и обосновать вывод о том, в каких случаях целесообразно применять алгоритм А, а в каких − алгоритм В.

|  |  |  |
| --- | --- | --- |
| Вариант | Вариант пары алгоритмов | Вариант эксперимента |
| 1 | 1 | 3.1 |
| 2 | 1 | 3.2 |
| 3 | 1 | 3.3 |
| 4 | 1 | 3.4 |
| 5 | 2 | 3.1 |
| 6 | 2 | 3.2 |
| 7 | 2 | 3.3 |
| 8 | 2 | 3.4 |
| 9 | 3 | 3.1 |
| 10 | 3 | 3.2 |
| 11 | 3 | 3.3 |
| 12 | 3 | 3.4 |
| 13 | 4 | 3.1 |
| 14 | 4 | 3.2 |
| 15 | 4 | 3.3 |
| 16 | 4 | 3.4 |