Министерство образования Республики Беларусь
Учреждение образования «Витебский государственный университет имени П.М. Машерова»
Кафедра информатики и информационных технологий

А.В. Кухарев, Е.А. Витько, А.А. Царев

АЛГОРИТМЫ НА ГРАФАХ

Методические указания к выполнению лабораторных работ по дисциплине «Теория графов»

Витебск
ВГУ имени П.М. Машерова
2016

УДК 519.17(076.5)
ББК 22.174.2я73
К95

Печатается по решению научно-методического совета учреждения образования «Витебский государственный университет имени П.М. Машерова». Протокол № 5 от 25.05.2016 г.

Авторы: преподаватель кафедры информатики и информационных технологий ВГУ имени П.М. Машерова, кандидат физико-математических наук **А.В. Кухарев;** доцент кафедры информатики и информационных технологий ВГУ имени П.М. Машерова, кандидат физико-математических наук **Е.А. Витько;** и.о. заведующего кафедрой информатики и информационных технологий ВГУ имени П.М. Машерова, кандидат физико-математических наук **А.А. Царев**

Р е ц е н з е н т ы :
профессор кафедры высшей алгебры и защиты информации БГУ, доктор физико-математических наук, профессор *Г.Е. Пунинский;*
заведующий кафедрой прикладной математики и механики ВГУ имени П.М. Машерова, кандидат физико-математических наук *С.А. Ермоченко*

Кухарев, А.В.

К95 Алгоритмы на графах : методические указания к выполнению лабораторных работ по дисциплине «Теория графов» / А.В. Кухарев, Е.А. Витько, А.А. Царев. – Витебск : ВГУ имени П.М. Машерова, 2016. – 38 с.

Издание содержит методические указания к выполнению лабораторных работ по дисциплине «Теория графов», контрольные вопросы, основные теоретические сведения из теории графов, а также справочную информацию по языку программирования Python и его библиотекам, предназначенным для работы с графами.

Предназначено для студентов дневной и заочной форм обучения по специальностям «Прикладная информатика (программное обеспечение компьютерных систем)» и «Программное обеспечение информационных технологий. Базы данных и программное обеспечение информационных систем».

УДК 519.17(076.5)
ББК 22.174.2я73

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ

$\{a_1, ..., a_n\}$ – множество элементов;

$[a_1, ..., a_n]$ – список, упорядоченная последовательность элементов произвольной природы;

(a, b) – пара элементов некоторого множества (упорядоченная либо неупорядоченная[1]);

$\min X$ ($\max X$) – минимальный (максимальный) элемент линейно упорядоченного множества X;

N – множество натуральных чисел;

$V(G)$ – множество вершин графа G;

$E(G)$ – множество дуг (ребер) графа G;

$\Gamma(v)$ – окрестность вершины v, т.е. множество вершин u (неориентированного) графа, таких, что существует ребро (v, u);

$\Gamma(V_1) := \bigcup_{v \in V_1} \Gamma(v)$, где $V_1 \subseteq V(G)$.

[1] Для удобства будем обозначать дуги и ребра графов одинаковым образом посредством (v_1, v_2), где v_1, v_2 – вершины орграфа либо неорграфа.

ВВЕДЕНИЕ

Целью лабораторных работ по дисциплине «Теория графов» является получение студентами-программистами навыков решения задач на графах с использованием современных высокоуровневых объектно-ориентированных языков программирования.

В методических указаниях даны задания к лабораторным работам, выполнение которых должно обеспечить закрепление студентами основных понятий теории графов и способствовать развитию навыков применения теоретических знаний к решению прикладных задач.

Реализовывать алгоритмы на графах в рамках данных лабораторных работ рекомендуется на языке программирования Python. Python – это высокоуровневый язык прикладного программирования. В языке Python реализованы основные классы для работы с множествами, последовательностями, матрицами и другими базовыми математическими абстракциями, что позволяет программисту сосредоточиться на решении практической задачи, отводя на задний план технические детали (например, вопросы выделения/освобождения памяти, программирования графики и т.п.). Кроме того, под Python существует множество расширений, включая библиотеки для работы с графами. В приложениях к изданию даны минимальные сведения о языке Python и его библиотеках, необходимые для выполнения лабораторных работ.

Также приводятся основные определения из теории графов, которые используются в данном издании. Определение графа дается в теоретико-множественном смысле.

Адресовано студентам дневной и заочной форм обучения по специальностям «Прикладная информатика (программное обеспечение компьютерных систем)» и «Программное обеспечение информационных технологий. Базы данных и программное обеспечение информационных систем».

ОБЩИЕ РЕКОМЕНДАЦИИ К ВЫПОЛНЕНИЮ ЛАБОРАТОРНЫХ РАБОТ

В описании к каждой лабораторной работе представлен один или несколько алгоритмом на графах, записанных на псевдокоде[2]. Необходимые определения из теории графов даны в приложении А.

В ходе выполнения лабораторной работы необходимо реализовать представленные алгоритмы на языке программирования Python, краткая справка по которому дана в приложении Б. Предполагается, что граф задан матрицей смежности либо матрицей инцидентности, которые требуется прочитать из файла на диске.

При реализации алгоритмов требуется использовать различные библиотеки (на свой выбор) языка Python для построения и вывода изображения графов на экран и/или сохранения их в файл. Обзор некоторых библиотек Python для работы с графами дан в приложении В.

ЛАБОРАТОРНАЯ РАБОТА № 1
Компоненты сильной связности

На псевдокоде алгоритм выделения компонент сильной связности (бикомпоненты) в орграфе $G = (V, E)$ может быть записан в следующем виде.

Input

V – множество вершин графа G;

S – матрица сильной связности графа G.

Begin

$S' := S$;

$i := 0$;

$V' := V$;

while $S' \neq \varnothing$ **do begin**

$i := i + 1$;

[2] *Псевдокод* – компактное описание алгоритма с использованием неформального языка с опусканием несущественных деталей. В настоящем пособие будем придерживаться паскалеподобного синтаксиса.

K_i := {$v \in V'$ | вершине v соответствует "1" в первой строке матрицы S'};

S' := матрица, получаемая из S' удалением строк и столбцов, соответствующих вершинам из K_i;

V' := $V' \setminus K_i$;

end

end.

Output

$\boldsymbol{K_j}$, j = 1,..., i – компоненты сильной связности графа G.

Задание. Написать программу, реализующую алгоритм поиска бикомпонент орграфа, заданного своей матрицей смежности. Использовать графические библиотеки языка Python для визуализации графа и его бикомпонент.

Контрольные вопросы

1. Что такое компонента сильной/слабой связности орграфа?
2. Какие существуют способы задания графов?
3. Как найти матрицу сильной связности графа, если задана его матрица смежности?
4. В чем заключается алгоритм поиска компонент сильной связности орграфа?

ЛАБОРАТОРНАЯ РАБОТА № 2
Обход вершин графа

Алгоритмы поиска (обхода) предназначены для систематического перечисления вершин неориентированного графа, в результате которого каждой вершине v графа (если он связный) приписывается некоторая метка $s(v)$.

Существует два основных способа обхода вершин графа – в глубину и в ширину. На практике эти алгоритмы чаще всего используются как составные части других алгоритмов.

Псевдокод поиска в глубину записывается следующим образом:

Input

$G = (V, E)$ – граф;

$p(v)$ – вершина-предшественник для вершины v;

u – начальная вершина.

Begin

$k := 0$; # номер шага

$s(u) := 0$; # $s(u)$ – метка вершины u;

$v^* := u$; # v^* – текущая вершина.

$V^* := V \setminus \{u\}$;

while $V^* \neq \varnothing$ **do begin**

if $V^* \cap \Gamma(v^*) \neq \varnothing$ **then do begin**

$k := k + 1$;

$v :=$ произвольная вершина из $V^* \cap \Gamma(v^*)$;

$s(v) := k$;

$p(v) := v^*$;

$V^* := V^* \setminus \{v\}$;

$v^* := v$;

end

else $v^* := p(v)$; # «возвращение назад»

end.

Output

$\{(v, s(v)) \mid v \in V\}$.

Поиск в ширину обеспечивает разбиение множества V вершин графа на уровни (классы C_k) в зависимости от их удаленности от начальной вершины. Псевдокод алгоритма:

Input

$G = (V, E)$ – граф;

u – начальная вершина.

Begin

$k := 0$; # номер шага

$s(u) := 0$;

$C_0 := \{u\}$;

$V^* := V \setminus C_0$;

while $V^* \neq \varnothing$ **do begin**

$k := k + 1$;

for each v **in** $V^* \cap \Gamma(C_{k-1})$ **do** $s(v) := k$;

$C_k := \{v \in V^* \mid s(v) = k\}$; # множество вершин с меткой k;

$V^* := V^* \setminus C_k$;

end.

Output

$\{ C_i \mid i = 0,..., k\}$.

Задания. 1) Написать программы, реализующие алгоритмы поиска в ширину и в глубину в графе, заданном матрицей смежности. 2) Выполнить проверку графа на связность. 3) С использованием алгоритма поиска в ширину найти радиус и диаметр графа, а также центр графа и его периферийные вершины.

Контрольные вопросы

1. В чем заключается задача обхода вершин графа?

2. В чем основное отличие алгоритма поиска в ширину от алгоритма поиска в глубину?

ЛАБОРАТОРНАЯ РАБОТА № 3
Нахождение минимального остова

Пусть задан связный нагруженный неориентированный граф $G = (V, E)$. Требуется найти его остовное дерево G^* минимального веса. Наиболее известными алгоритмами нахождения минимального остова графа являются алгоритмы Краскала и Прима.

Псевдокод алгоритма Краскала:

Input

$G=(V,E)$ – граф.

Begin

e := ребро минимального веса из E;

$T := \{e\}$;

$E' := E \setminus \{e\}$;

while $E' \neq \varnothing$ **do**

begin

e' := ребро минимального веса из E';

$T := T \cup \{e'\}$;

$E' := (E' \setminus T) \setminus \{e \in E \mid T \cup \{e\}$ содержит цикл$\}$;

end

end.

Output

$G^* = (V, T)$ – минимальный остов.

Псевдокод алгоритма Прима:

Input

$G = (V, E)$ – граф.

Begin

u := любая вершина из V;

$V_T := \{u\}$;

while $V_T \neq V$ **do**

begin

$E^* := \{ (u,v) \in E \mid u \in V_T, v \in V \setminus V_T\}$;

(u,v) := ребро минимального веса из E^*;

$T := T \cup \{(u,v)\}$;

$V_T := V_T \cup \{u,v\}$;

end

end.

Output

$G^* = (V_T, T) = (V, T)$ – минимальный остов.

Примечание. В этом коде множество E^* представляет собой множество ребер графа G, инцидентных ровно одной вершине из дерева T.

Задание. Написать программы для нахождения минимального остова по алгоритмам Краскала и Прима в графе, заданном матрицей смежности. Изобразить граф и его минимальный остов.

Контрольные вопросы

1. Что такое дерево, остовное дерево?
2. В чем основное различие между Краскала и Прима?

ЛАБОРАТОРНАЯ РАБОТА № 4

Нахождение эйлерова цикла

Путь P в графе $G = (V, E)$ можно представить в виде упорядоченной последовательности ребер $e_1e_2...e_n$, где $e_i \in E$, причем e_i и e_{i+1} ($i = 1,..., n$-1) должны иметь общую вершину. Для двух заданных путей $P_1 = e_1e_2...e_n$ и $P_2 = e_{n+1}e_{n+2}...e_m$ (где $m > n$) графа G, таких что $e_n = (u, v)$ и $e_{n+1} = (v, w)$, определим операцию конкатенации путей следующим образом: $P_1 + P_2 := e_1e_2...e_ne_{n+1}e_{n+2}...e_m$. Через $P(u,v)$ будем обозначать путь с началом в вершине u и концом в вершине v (предполагается, что направление обхода пути P зафиксировано).

Псевдокод поиска эйлерового пути в графе:

Input

$G = (V, E)$ – граф.

Begin

$P := \varnothing$; # это будет искомый эйлеров цикл

v := произвольная вершина из V;

$a := v$; # это будет «начальная» вершина всего пути P

while $E \neq \varnothing$ **do**

begin

if $P \neq \varnothing$ **then** v := вершина из V, т.ч. $\exists e \in E$: e инцидентно v;

$c := v$; # на первой итерации будет $a = c = v$

$P' := \varnothing$; # внутренний цикл

repeat

u := вершина из V, т.ч. $(v,u) \in E \backslash P'$; # u – конец ребра

$P' := P' + [(v,u)]$; # формируем путь (цикл)

$E := E \setminus \{(v,u)\}$; # вычеркиваем ребро

$v := u$; # теперь это начало следующего ребра

until $u \neq v$;

if $P \neq \varnothing$ **then** $P := P(a,c) + P' + P(c,a)$;

else $P := P'$;

end

end.

Output

P – эйлеров цикл.

Задание. 1) Написать программу для поиска эйлерова цикла в графе, заданном матрицей смежности. Программа должна выполнять предварительную проверку на существование эйлерова цикла, и если цикл существует, то (визуально) выделять его на графе. 2) Модифицировать алгоритм для нахождения эйлеровой цепи.

Контрольные вопросы

1. Какой цикл (цепь) называется эйлеровом?
2. Какие существуют критерии существования эйлерового цикла в графе?
3. В чем заключается алгоритм поиска эйлера цикла в графе?

ЛАБОРАТОРНАЯ РАБОТА № 5
Поиск кратчайшего пути. Алгоритм Дейкстры

Пусть дан нагруженный граф $G = (V,E)$ с некоторой матрицей весов $W = [w(v_i,v_j)]_{ij}$, где $v_i, v_j \in V$. Алгоритм Дейкстры позволяет найти кратчайший путь (т.е. путь минимального веса) из заданной вершины A во все остальные вершины графа G. В результате работы алгоритма будет найден вес кратчайшего пути, а также построен сам путь (точнее, один из таких путей, если их несколько). Этот путь будет представлять собой список $P(v)$ вершин графа, где первая вершина есть A, а последняя – v.

Input

$G = (V, E)$ – граф;

$A \in V$ – начальная вершина.

Begin

for each v **in** V **do** begin

$P(v) := [A]$;

if $v \in \Gamma(A)$ **then**

$d(v) := w(A, v)$;

else $d(v) := \infty$;

end

$V^* := \{A\}$;

$V^* \subseteq V$ – множество отмеченных вершин

while $V^* \neq V$ **do**
begin
u := вершина из $V \setminus V^*$ с минимальным значением $d(u)$;
$V^* := V^* \cup \{u\}$;
for each v **in** $\Gamma(u) \setminus V^*$ **do begin**
$d' := d(u) + w(u, v)$;
if $d' < d(v)$ **then begin**
$d(v) := d'$;
$P(v) := P(u) + [v]$;
end
end
end
end.

Output

$d(v)$, $v \in V\setminus\{A\}$ – искомые расстояния;
$P(v)$ – кратчайший путь из вершины A до вершины v.

Задание. Реализовать алгоритм Дейкстры для поиска кратчайшего пути в нагруженном графе с заданной матрицей весов. Построить граф и выделить кратчайший путь.

Контрольные вопросы

1. Какой граф называется нагруженным?
2. Для чего предназначен и в чем заключается алгоритм Дейкстры?

ЛАБОРАТОРНАЯ РАБОТА № 6
Поиск кратчайшего пути. Алгоритм Форда-Беллмана

В отличие от алгоритма Дейкстры, в алгоритме Форда-Беллмана происходит поиск кратчайших путей между всеми парами вершин нагруженного графа G. В результате работы алгоритма будет получена матрица кратчайших путей $D = (d_{ij})$, а также матрица предшественников $P = (p_{ij})$, позволяющая восстановить эти пути.

Input

$G = (V,E)$ – граф;
$W = (w_{ij})$ – матрица весов графа G.

Begin

$k := 1$; # номер шага

$D^{(1)} := W$;

for i, j **from** 1 **to** n **do begin**

if $w_{ij} = \infty$ **then** $p_{ij}^{(1)} := 0$;

else $p_{ij}^{(1)} := i$;

end

repeat

$k := k + 1$;

for i, j **from** 1 **to** n **do** $d_{ij}^{(k)} := \min\limits_{1 \le m \le n} \{d_{im}^{(k-1)} + w_{mj}\}$;

for i, j **from** 1 **to** n **do begin**

if $d_{ij}^{(k)} = d_{ij}^{(k-1)}$ **then** $p_{ij}^{(k)} := p_{ij}^{(k-1)}$;

else if $d_{ij}^{(k)} = d_{im}^{(k-1)} + w_{mj}$ **then** $p_{ij}^{(k)} := m$;

end

until $D^{(k)} = D^{(k-1)}$;

Output

$D := D^{(k)} = (d_{ij}^{(k)})$ – матрица кратчайших расстояний;

$P := P^{(k)} = (p_{ij}^{(k)})$ – матрица предшественников.

Задание. Реализовать алгоритм Форда-Беллмана для поиска кратчайшего пути в нагруженном графе с заданной матрицей весов. Построить граф и выделить кратчайший путь.

Контрольные вопросы

1. В чем заключается отличие алгоритма Форда-Беллмана от алгоритма Дейкстры?

2. Как определяется матрица предшественников, матрица кратчайших расстояний?

ЛАБОРАТОРНАЯ РАБОТА № 7

Построение наибольшего паросочетания в двудольном графе

Пусть $G(X, Y, E)$ – двудольный граф, в котором выделено паросочетание P. На первом этапе алгоритма строится орграф $G^*(X, Y, E^*)$, вершины которого совпадают с вершинами графа G, а ориентация дуг

зависит от того, принадлежит ли соответствующее ребро графа G паросочетанию P. В процессе выполнения цикла **while** на каждой итерации количество ребер в паросочетании P увеличивается на одно до тех пор, пока не будет получено наибольшее паросочетание.

Input

$G = (X, Y, E)$ – двудольный двудольный граф;

$P \subseteq E$ – некоторое паросочетание в G;

$X_P \subseteq X$ и $Y_P \subseteq Y$ – вершины, инцидентные ребрам паросочетания P.

Begin

*# построение орграфа G**

for each (x, y) **in** $\{(v_X, v_Y) \mid v_X \in X, v_Y \in Y\}$ **do begin**

if $(x, y) \in P$ **then** $E^* := E^* \cup \{(y, x)\}$; *# дуга из Y в X*

else $E^* := E^* \cup \{(x, y)\}$; *# дуга из X в Y*

end

 # построение наибольшего паросочетания P

while по алгоритму поиска в ширину найден путь в орграфе G^* из какой-либо вершины $X \setminus X_P$ в какую-либо вершину $Y \setminus Y_P$ **do begin**

$(u, w_1, w_2, ..., w_k, v)$:= найденный пусть, где $u \in X \setminus X_P$, $v \in Y \setminus Y_P$;

$X_P := X_P \cup \{u\}$;

$Y_P := Y_P \cup \{v\}$;

$E^* := (E^* \setminus \{(w_1, w_2), (w_3, w_4), ..., (w_{k-1}, w_k)\}) \cup$

$\cup \{(u, w_1), (w_2, w_3), ..., (w_k, v)\}$;

end

P := множество ребер, соответствующие дугам из E^*.

Output

P – наибольшее паросочетание.

Задание. Написать программу для поиска наибольшего паросочетания в двудольном графе.

Контрольные вопросы

1. Какой граф называется двудольным? Какие существуют критерии того, что заданный граф является двудольным?

2. Что такое паросочетание, наибольшее паросочетание?

ЛАБОРАТОРНАЯ РАБОТА № 8

Раскраска графа

Задача раскраски графа заключается в следующем. Требуется каждой вершине v графа G сопоставить некоторое натуральное число $\varphi(v)$, называемое *цветом* вершины, так, чтобы смежные вершины графа имели разные цвета. При этом необходимо использовать минимально возможное число цветов. Полученную таким образом функцию φ: $V(G) \to \mathbf{N}$ будем называть *раскраской* графа.

Самыми простыми алгоритмами раскраски графа являются жадный алгоритм и алгоритм последовательного раскрашивания.

Псевдокод жадного алгоритма раскрашивания графа:

Input

$G = (V, E)$ – граф.

Begin

$S := V$; # $S \subseteq V$ – множество неокрашенных вершин;

$k := 0$;

for each v **in** V **do** $\varphi(v)$:= UNDEFINED;

while $S \neq \varnothing$ **do**

begin

v := любая вершина из S;

$k := k+1$;

$C_k := \{v\}$;

while $S \setminus (C_k \cup \Gamma(C_k)) \neq \varnothing$ **do begin**

u := любая вершина из $S \setminus (C_k \cup \Gamma(C_k))$;

$C_k := C_k \cup \{u\}$;

end

for each v **in** C_k **do** $\varphi(v) := k$;

$S := S \setminus C_k$;

end

end

Output

$\varphi(v_i)$, $i = 1,...,|V|$, – раскраска вершин графа.

Псевдокод алгоритма последовательного раскрашивания графа:

Input

$G = (V,E)$ – граф.

begin

$n := |V|$;

for each v **in** V **do** $S(v) := \{1, 2,..., n\}$;

$S(v)$ – множество «допустимых» цветов для данной вершины v;

for k **from** 1 **to** $n - 1$ **do**

begin

$\varphi(v_k) := \min S(v_k)$;

for each v **in** $\Gamma(v_k)$ **do** $S(v) := S(v) \setminus \{\varphi(v_k)\}$;

end

$\varphi(v_n) := \min S(v_n)$;

end

Output

$\varphi(v_i)$, $i = 1,...,|V|$, – раскраска вершин графа.

Задание. Реализовать жадный алгоритм и алгоритм последовательного раскрашивания графа. Использовать возможности графических библиотек под Python для раскраски вершин графа в разные цвета.

Контрольные вопросы

1. В чем заключается задача раскраски графа?

2. В чем особенности жадного алгоритма и алгоритма последовательного раскрашивания?

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Оре, О. Теория графов / О. Оре. – М.: Наука, 1968. – 352 с.

2. Емеличев, В.А. Лекции по теории графов / В.А. Емеличев, О.И. Мельников, В.И. Сарванов, Р.И. Тышкевич. – М.: Наука, 1990. – 384 с.

3. Diestel, R. Graph Theory / R. Diestel. – Heidelberg: Springer-Verlag, 2005. – 431 p.

4. Хаггарти, Р. Дискретная математика для программистов / Р. Хаггарти. – 2-е изд. – Пер. с англ. – М.: Техносфера, 2005. – 400 с.

5. Храмова, Т.В. Лекции по теории графов. Учебное пособие / Т.В. Храмова. – Новосибирск: СибГУТИ, 2011. – 98 с.

6. Носов, В.И. Элементы теории графов. Учебное пособие / В.И. Носов, Т.В. Бернштейн, Н.В. Носкова, Т.В. Храмова. – Новосибирск, 2008. – 107 с.

7. Домнин, Л.Н. Элементы теории графов / Л.Н. Домнин. – Пенза, 2004. – 139 с.

8. Лутц, М. Изучаем Python / М. Лутц. – 4-е изд. – Пер. с англ. – СПб.: Символ-Плюс, 2011. – 1280 с.

ПРИЛОЖЕНИЯ

ПРИЛОЖЕНИЕ А

Основные определения теории графов

Ориентированный граф (орграф[3]*)* – это пара $G = (V, E)$, где V – произвольное множество, и $E \subseteq V \times V$ – бинарное отношение на V. Элементы из V называются *вершинами*, а элементы из E – *дугами* графа.

Неориентированный граф (неорграф) – это пара $G = (V, E)$, где V – произвольное множество, и $E \subseteq V^{(2)}$, $V^{(2)}$ – множество всех двухэлементных подмножеств множества V. Элементы из V называются *вершинами*, а элементы из E – *ребрами*.

Нагруженный (*взвешенный*) граф – граф, в котором каждой дуге (ребру) приписано вещественное число, называемое её *весом*.

Подграф графа $G = (V, E)$ – это любой граф $H = (V_1, E_1)$, такой, что $V_1 \subseteq V$ и $E_1 \subseteq E$.

Маршрут – последовательность дуг (ребер) графа, в котором начало последующей дуги совпадает с концом предыдущей.

Длина маршрута – число дуг (ребер) в маршруте.

Расстояние $d(v, u)$ от вершины v до вершины u – это минимальная длина среди длин всех маршрутов из вершины v в вершину u.

Удалённость (*эксцентриситет*) $\varepsilon(v)$ вершины v графа G – это наибольшее из расстояний от вершины v до всех остальных вершин графа G.

Радиус графа – это наименьшая из удалённостей его вершин.

Диаметр графа – это наибольшая из удалённостей его вершин.

Центр графа – вершина, удалённость которой равна радиусу графа.

Периферийная вершина – вершина, удалённость которой равна диаметру графа.

Путь (*цепь*) – маршрут, все дуги (ребра) которого различны.

Простой путь (*простая цепь*) – путь (цепь), все вершины которого различны.

Цикл – замкнутый путь (замкнутая цепь[4]).

[3] Мы будем рассматривать только графы без кратных дуг и без петель. Многие определения формулируются одинаково для ориентированных и неориентированных графов.

[4] Часто замкнутую цепь в неорграфе называют *контуром*, однако для упрощения терминологии будем использовать термин «цикл» и в это в случае.

Эйлеров цикл – цикл, который содержит все дуги (ребра) графа ровно один раз.

Связный неорграф – неорграф, в котором любые две вершины могут быть соединены маршрутом.

Дерево – связный неорграф без циклов.

Остов (остовное дерево) графа – любой подграф, который содержит все вершины графа и является при этом деревом.

Связный орграф – орграф, в котором для любой пары вершин u, v существует путь из u в v, либо путь из v в u.

Сильно связный орграф – орграф, любые две вершины которого *достижимы* друг из друга (т.е. для каждой пары вершин существует путь как из первой вершины во вторую, так и обратный путь).

Слабо связный орграф – орграф, который не является связным, но который становится связным неорграфом при замене всех дуг ребрами.

Бикомпонента (компонента сильной связности) орграфа – максимальный по включению сильно связный подграф.

Смежные вершины – вершины графа, для которых существует дуга (ребро), их соединяющая.

Матрица смежности графа – это матрица $A = (a_{ij})$, такая, что $a_{ij} = 1$, если существует дуга (ребро) из вершины v_i в вершину v_j, и $a_{ij} = 0$ в противном случае.

Инцидентность – отношение, в котором находятся ребро (дуга) $e = (x, y)$ и вершина v графа тогда и только тогда, когда $x = v$ либо $y = v$.

Матрица инцидентности графа – это матрица $B = (b_{ij})$, строки которой соответствуют вершинам, а столбцы – дугам (ребрам) графа. Для орграфа: элемент матрицы $b_{ij} = 1$, если i-ая вершина является началом j-ой дуги; и $b_{ij} = -1$, если i-ая вершина является концом j-ой дуги, и $b_{ij} = 0$ во всех остальных случаях. Для неорграфа: $b_{ij} = 1$, если i-ая вершина инцидентна j-му ребру, и $b_{ij} = 0$ в противном случае.

Матрица достижимости орграфа – матрица $R = (r_{ij})$, такая, что $r_{ij} = 1$, если существует путь из вершины v_i в вершину v_j, и $r_{ij} = 0$ в остальных случаях.

Матрица сильной связности орграфа – матрица $S = (s_{ij})$, такая, что $s_{ij} = 1$, если существует путь из вершины v_i в вершину v_j и существует обратный путь из v_i в v_j, и $s_{ij} = 0$ в противном случае.

Паросочетание в неорграфе – произвольное подмножество попарно несмежных ребер (т.е. ребер без общих вершин).

Наибольшее паросочетание – паросочетание с наибольшим числом ребер среди всех паросочетаний графа.

Совершенное паросочетание – паросочетание, покрывающее все вершины графа.

Двудольный граф – граф, множество вершин которого можно разбить на два подмножества попарно несмежных вершин.

ПРИЛОЖЕНИЕ Б

Справка по языку программирования Python

Общая характеристика языка. Python – кросплатформенный высокоуровневый интерпретируемый язык прикладного программирования. Его синтаксис минималистичен и интуитивно понятен, что облегчает его изучение. В тоже время это достаточно мощный язык программирования благодаря наличию огромного количества библиотек и расширений под него.

Python – сравнительно молодой и активно развивающийся язык. Первая версия вышла в 1991 г. В настоящее время параллельно используются две версии языка Python: вторая и третья. В этом пособии примеры даны для третьей версии.

В настоящее время основными сферами применения Python являются веб-разработка и научные вычисления. Python может составить конкуренцию коммерческому продукту Matlab благодаря наличию модулей расширения NumPy, SciPy, MatPlotLib и многих других. В рамках настоящего предмета нас в первую очередь интересуют возможности Python для программирования задач на графах.

Python полностью поддерживает объектно-ориентированную парадигму, а также частично поддерживает функциональное программирование. Однако в данном пособии мы не будем касаться вопросов разработки собственных классов, а также рассматривать работу с исключениями. Для наших целей вполне достаточно тех классов, которые уже реализованы в Python.

Это краткое введение в язык Python рассчитано на читателей, имеющих определенный опыт программирования на одном или нескольких объектно-ориентированных языках программирования (C++,

C#, JAVA, Delphi и т.п.). Поэтому некоторые понятия будут даваться в сравнении с этими языками.

Начало работы с Python. Начинать осваивать Python лучше с примеров. Так, самая простая программа, выводящая на консоль фразу «Hello World!», на языке Python состоит всего из одной строчки

```
print('Hello World!')
```

В терминологии интерпретируемых языков такая программа называется *сценарием*. Достаточно сохранить этот сценарий в файл под именем, скажем, prog.py, а затем выполнить его с помощью интерпретатора языка Python. Будет получен следующим результат:

```
$ python3 prog.py
Hello World!
```

Если в консоли набрать `python3` без параметров, то попадем в так называемый *интерактивный режим*, о чем свидетельствуют приглашение ввода `>>>`. Этот режим удобен на начальной стадии изучения языка, поскольку позволяет пошагово выполнять инструкции и тут же наблюдать за результатом их выполнения:

```
$ python3
Python 3.4.2 (default, Oct  8 2014, 13:14:40)
[GCC 4.9.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Interactive Mode")
Interactive Mode
```

Комментарии могут быть добавлены в код программы после символа решетки «#»:

```
print('Hello World!') # это комментарий
```

Инструкции в Python заканчиваются переводом строки. При этом нет необходимости ставить в конце точку с запятой (как, например, в C++), хотя ее постановка не будет восприниматься интерпретатором как ошибка. Точка с запятой используется в том случае, если нужно разместить несколько инструкций в одной строке, например

```
>>> x = 2; y = 3; print(x + y)
5
```

Типы данных. Python – язык со строгой динамической типизацией. Строгая типизация означает, что переменная в любой момент времени

имеет точно определенный тип. Тип переменной можно определить с помощью встроенной функции `type`:

```
>>> X = 10
>>> type(X)
<class 'int'>
>>> X = 'Hello'
>>> type(X)
<class 'str'>
```

Здесь знак «равно» (`=`) означает оператор присваивания. Это аналогично синтаксису языков C/C++/JAVA. Однако в отличие от этих языков, в Python нет необходимости указывать тип переменной. Он определяется автоматически в ходе выполнения программы (это так называемая *динамическая типизация*). Более того, переменная может изменить свой тип в ходе выполнения программы, если ей будет присвоено новое значение другого типа. Как говорят, в языке Python «важнее значение, а не имя переменной». Удобство такого подхода проявится, например, при написании функции. Таким образом, имена переменных в Python – это только ссылки на объекты. Но они не хранят информацию о типе объекта.

К основным встроенным типам данных языка Python относятся числа, строки, множества, списки, кортежи и словари. Рассмотрим их подробнее (в скобках указан тип данных, возвращаемый функцией `type`):

- *Числовые* типы, которые включают целые (`int` и `long`), вещественные (`float`) и комплексные (`complex`) числа.
- *Строка* (`str`) – последовательность символов Юникода. Строки заключаются в кавычки либо в апострофы (оба варианта равноценны). Например, `"Python"` или `'Python'`.
- *Множество* (`set`). Пример множества: `{1,2,3}`.
- *Список* (`list`) – упорядоченная последовательность значений произвольных типов. Пример списка: `[1,2,3]`.
- *Кортеж* (`tuple`) – аналогичен списку, только неизменяемый. Пример кортежа: `(1,2,3)`.
- *Словарь* (`dict`) – неотсортированная колекция элементов, доступ к которым осуществляется по ключу. Пример словаря: `{1:'a', 2:'b'}`. По большому счету, словарь в Python – это список кортежей.
- *Логический* тип (`bool`) – тип данных, переменные которого могут принимать только два значения: `True` и `False`.

В Python все базовые типы реализованы в виде объектов, причем для них наиболее важные операторы уже перегружены. Поэтому работать с базовыми типами можно двумя способами: либо вызывать методы объектов по имени, либо использовать перегруженные операторы.

Продемонстрируем сказанное на примере класса `int`, реализующего целые числа. В этом классе перегружены базовые операторы `+`, `-`, `*`, `/` и `**`, представляющие собой арифметические операции (две звездочки `**` означают возведение в степень). Например, следующий код

```
>>> x = 2 + 3
```

может быть переписан в эквивалентном виде с использованием метода `__add__()` класса `int`:

```
>>> x = (2).__add__(3)
```

Но, конечно, так никто не пишет.

Узнать полный список методов любого класса можно посредством функции `dir`, указав нужный класс в качестве аргумента этой функции. Например,

```
>>> dir(int)
```

Для получения более детальной справки о классе и его методах используйте функцию `help`:

```
>>> help(int)
>>> help(int.__add__)
```

Среди полезных возможностей Python следует отметить возможность работы с комплексными числами (тип `complex`). Мнимая единица задается как `1j` либо `1J`. Кроме того, существует модуль `fractions`, который поддерживает работу с рациональными дробями.

Списки. Списки в Python – это аналог массивов из других языков программирования. Поэтому они используются довольно часто. Списки в Python задаются в виде заключенной в квадратные скобки последовательности элементов, перечисленных через запятую. При этом элементы списка могут иметь любой тип (не обязательно одинаковый), в частности, они могут являться тоже списками. Рассмотрим некоторые часто используемые операции над списками.

Получить элемент списка можно, указав номер элемента в квадратных скобках после имени списка (нумерация элементов списка начинается с нуля):

```
L = [1, 2, 'A', True, [3, 4]]
>>> L[0]     # первый элемента списка
1
>>> L[4][0]  # извлекаем элемент вложенного списка
3
```

Выполнить конкатенацию (объединение) двух списков можно с помощью перегруженного оператора + («плюс»):

```
>>> [1, 2] + [5, 6]
[1, 2, 5, 6]
```

Кроме того, класс `list` имеет набор различных методов для вставки и удаления элементов из списка: append(), insert(), remove(), pop(). Имеются также методы для сортировки элементов списка: sort() и reverse().

Например, метод insert() используется для вставки нового элемента в середину (или начало) списка. Первый аргумент этого метода указывает позицию, куда нужно вставить элемент, а вторым аргументом передается сам элемент:

```
>>> L = [1, 2]
>>> L.insert(1, 3)
>>> L
[1, 3, 2]
```

Узнать длину списка можно с помощью функции `len`:

```
>>> len([1,2,3])
```

Сделаем одно важное замечание относительно операции присваивания (=) на примере работы со списками. Эта операция не производит копирование списков, а только создает еще одну ссылку на тот же самый список (т.е. хранящийся в той же области памяти). Это сделано для экономии ресурсов, однако может привести к неожиданным результатам при неправильном использовании. Рассмотрим следующий пример:

```
>>> L = [1, 2, 3]
>>> S = L
>>> L[0] = 9   # изменим первый элемент списка L
>>> L
[9, 2, 3]
>>> S
[9, 2, 3]
```

Обратите внимание, что в списке S первый элемент тоже изменился, поскольку на самом деле переменные S и L ссылаются на один и тот же список.

Если же нам нужно сделать реальную копию списка, то для этих целей существует метод copy():

```
>>> L = [1, 2, 3]
>>> S = L.copy()
>>> L[0] = 9
>>> L
[9, 2, 3]
>>> S       # теперь все в порядка
[1, 2, 3]
```

Это замечание справедливо и для остальных типов данных языка Python. В общем случае для операции присваивания действуют следующие правила:

– Инструкция присваивания всегда создает ссылку на объект (но не создает копии объектов).

– Переменные создаются при первом присваивании.

– Перед использованием переменной ей должно быть присвоено значение.

Множества. В отличие от списков, множества не могут содержать повторяющиеся элементы. Для работы с множествами предусмотрены методы union(), intersection(), difference(), issubset(), смысл которых понятен из названия. Посмотрим, как это выглядит на деле:

```
>>> A = {1, 2}
>>> B = {2, 3}
>>> A.union(B)
{1, 2, 3}
>>> A.intersection(B)
{2}
```

Те же операции можно записать в более компактной форме, используя перегруженные операторы:

```
>>> A | B    # объединение множеств
{1, 2, 3}
>>> A & B    # пересечение множеств
{2}
```

Также имеются операции нахождения разности множеств (`A-B`) и симметрической разности (`A^B`) множеств. Мощность множества можно узнать с помощью функции `len`.

Кроме того, существуют операции над множествами, которые возвращают логическое значение истина или ложь. К ним относится операция сравнения двух множеств (`A==B`), проверка вложенности одного множества в другое (`A<=B`), принадлежность элемента множеству (`x in A`) и другие.

Строки. Строки в Python – это не просто массив символов. Они реализованы в виде отдельного класса `str`. Однако операции над строками очень похожи на операции над списками. Например, возможна конкатенация строк (оператор `+`), получение символа по индексу (оператор `[]`), получение длины строки (функция `len`), повторение строки (оператор *) и другие. Следующий пример демонстрирует работу со строками.

```
>>> s = 'str'+'ing' # конкатенация
>>> s
'string'
>>> s[2]
'r'
>>> len(s)
6
```

Операторы сравнения. Операторы сравнения сравнивают между собой два элемента одного класса и в качестве результата возвращают константы `True` либо `False`. Основные операторы сравнения следующие: равно (`==`), не равно (`!=`), меньше (`<`), больше (`>`), не меньше (`>=`), не больше (`>=`). Например, для числовых типов получим

```
>>> 5 <= 3
False
```

Кроме того, операторы сравнения перегружены для некоторых других (не числовых) типов. Например, как уже было отмечено выше, применительно к множествам операторы `<` и `>` выполняют проверку на включение:

```
>>> {1,2,3} > {1,3}
True
```

Логические операции. Для переменных логического типа `bool` определены следующие операторы: `and`, `or` и `and`. Рассмотрим простой пример использования этих операторов:

```
>>> (not 1==2) or False
True
```

Преобразование типов данных. Язык Python допускает неявное преобразование типов. Например, если сложить целое число (тип `int`) с вещественным (тип `float`), то результатом будет число типа `float`:

```
>>> type(3 + 4.5)
<class 'float'>
```

Явное преобразование из одного типа в другой имеет следующую форму: `тип(значение)`. Рассмотрим пример преобразования строки в вещественное число:

```
>>> float('12'+'3')
123.0
```

Еще один полезный пример – преобразования строки в список символов:

```
>>> list('string')
['s', 't', 'r', 'i', 'n', 'g']
```

Блоки, циклы и ветвления. Оператор ветвления в Python представлен условной инструкцией `if`. Например,

```
if 2 > 1:
    print('условие выполнено')
```

В Python отсутствует оператор множественного выбора (как, например, switch/case в C++). Вместо этого используется полная конструкция `if/elif/else`, общая форма которой имеет следующий вид:

```
if <условие1>:
    <блок1>
elif <условие2>:
    <блок2>
else <условие3>:
    <блок3>
```

Для реализации циклов в Python присутствуют стандартные инструкции `for` и `while`. Инструкция `for` всегда используется с ключевым словом `in` и предназначена для обхода всех элементов

множества, списка или любой другой последовательности. Пример использования:

```
for x in {3, 2, 1}:
    print(x)
```

Стандартная форма цикла `while` следующая:

```
while <условие>:
    <блок_кода>
```

Совместно с инструкциями циклов могут использоваться стандартные инструкции `break` и `continue`, а также менее стандартные `pass` и `else`. Инструкция `pass` – это пустая инструкция, которая ничего не делает. Инструкция `else` располагается после цикла и всегда выполняется в том случае, если цикл завершается обычном способом (без прерывания с помощью `break`).

Сделаем важное замечание относительно блоков кода в языке Python. В отличие от многих других языков программирования, в Python отсутствуют какие-либо ключевые слова или конструкции для выделения блока кода (такие как **begin...end** в языке Pascal или фигурные скобки в C++). Вместо этого используются горизонтальные отступы (табуляции либо пробелы). В качестве примера рассмотрим следующий сценарий:

```
for x in [1, 2]:
    print('код внутри цикла')
print('код вне цикла')
```

В этом примере 2-ая строка кода входит в тело цикла `for` и поэтому строка «код внутри цикла» будет напечатана дважды, а инструкция 3-ей строки расположена за пределами цикла, и поэтому строка «код вне цикла» будет напечатана только один раз.

Для формирования последовательностей чисел с заданным шагом существует функция `range(`*start*, *stop*, *step*`).` Если шаг *step* > 0, то эта функция вернет возрастающую последовательность чисел, удовлетворяющих условию $start \le start + i \cdot step < stop$, где i – целое неотрицательное число. Эту последовательность затем можно преобразовать, например, в список:

```
>>> list(range(0, 10, 2))
[0, 2, 4, 6, 8]
```

Поэтому инструкция `range` часто используется в циклах, например

```
for x in range(0, 10, 2):
     print(x)
```

Эта функция также применяется для генерации списков (или других последовательностей) по заданной формуле, что демонстрирует следующий пример:

```
>>> [x**2 for x in range(0, 10, 2)]
[0, 4, 16, 36, 64]
```

Функции. Как и в большинстве языков программирования, в Python можно определять собственные функции. Для создания функции предназначена инструкция `def`. Ниже показан пример определения функции:

```
def sum(x,y):
     return x+y
```

По умолчанию все переменные внутри функции локальные, т.е. находятся в локальной области видимости. Это вполне естественный подход. Однако есть возможность обратиться и к глобальным переменным, объявленным вне функции. Для этого имеются ключевые слова `global` и `nonlocal`.

Еще одни способ задания функций – использование лямбда-нотации:

```
>>> sum = lambda x,y: x+y
>>> sum(2,3)
5
```

Модули. До сих пор рассматривались только функции и классы, входящие непосредственно в ядро языка Python. Их использование не требует подключения каких-либо внешних библиотек. Однако для расширения функциональности возникает необходимость подключения (импортирования) внешних модулей. Для этого служит ключевое слово `import`.

Например, модуль `math` содержит набор математических функций (`sqrt`, `abs`, `exp`, `log` и различные тригонометрические функции). Чтобы иметь возможность использовать эти функции в своей программе, необходимо выполнить импорт модуля `math`:

```
>>> import math
>>> math.pi              # число пи
```

```
3.141592653589793
>>> math.cos(math.pi)  # вычисляем косинус числа пи
-1.0
```

При импорте модуля можно задать для него псевдоним (для удобства) с помощью ключевого слова `as`, что позволит затем использовать псевдоним при обращении к атрибутам модуля. Например,

```
>>> import math as m
>>> m.cos(m.pi)
-1.0
```

Часто бывает полезно импортировать не весь модуль, а лишь отдельные его атрибуты (имена функций, классов и т.п.). Для этого служит инструкция `from`. Кроме того, после такого импорта для вызова функции достаточно указывать только её имя (без имени модуля):

```
>>> from math import cos, pi
>>> pi
3.141592653589793
>>> cos(pi)
-1.0
```

Существуют специальная форма инструкции `from`, которая позволяет импортировать сразу все атрибуты модуля:

```
>>> from math import *
```

Однако такой подход не всегда оправдан, так как приводит к «замусориванию» пространства имен.

Фактически модули в Python представляют собой просто файлы с исходным кодом либо байт-кодом (причем не обязательно написанных на языке Python). Несколько модулей могут быть помещены в отдельный каталог. Такой каталог называет *пакетом*. Тогда полный путь к функции будет выглядеть следующим образом: `пакет.модуль.функция`.

Ввод-вывод. Для считывания строки с консоли используется функция `input`, а для вывода на консоль – функция `print`. Например,

```
>>> s = input() # ввод строки с консоли
5
>>> x = int(s)  # преобразование в числовую форму
>>> y = x**2
>>> print(y)    # вывод на консоль
25
```

Чтение/запись данных из файла осуществляется с помощью методов `read()` и `write()` соответственно. Перед началом работы с файлом его

необходимо открыть с помощью функции `open()`, а после окончания работы – закрыть посредством `close()`. Типичный пример работы с файлом (в текстовом режиме):

```
>>> f = open('newfile.txt','w') # режим записи
>>> f.write('Hello')
5
>>> f.close()
>>> f = open('newfile.txt','r') # режим чтения
>>> f.read()
'Hello'
>>> f.close()
```

ПРИЛОЖЕНИЕ В

Библиотеки языка Python для работы с графами

В этом приложении дан обзор некоторых полезных инструментов, позволяющих упростить программирование задач на графах. Под Python существует несколько библиотек для работы с графами. Основные из них – это NetworkX, Igraph и Graph-tool. Первые две библиотеки просты в освоении, однако их возможности ограничиваются в основном визуализацией графов. Библиотека Graph-tool имеет несколько больше возможностей, в частности в ней реализованы некоторые базовые алгоритмы на графах.

Сначала кратко рассмотрим библиотеку NumPy, которая будет использована при работе с матрицами смежности графов, и библиотеку двумерной графики Matplotlib.

NumPy. Пакет модулей NumPy расширяет возможности языка Python и предназначен для работы с матрицами (класс `matrix`), а также обеспечивает поддержку больших многомерных массивов (класс `ndarray`). Эта библиотека не входит в стандартный набор пакетов, поставляемый с интерпретатором Python, и поэтому требует отдельной установки.

Следующий пример демонстрирует создание матрицы размера 2 на 2 и ее вывод на консоль:

```
>>> import numpy as np
>>> A = np.matrix([[1,2],[3,4]])
>>> print(A)
```

```
[[1 2]
 [3 4]]
```

Прокомментируем этот код. Во второй строчке происходит создание экземпляра класса `matrix` (из пакета `numpy`), причем в конструктор класса передается стандартный «питоновский» список, из которого формируется матрица A.

После того, как матрица создана, над ней можно производить различные операции. Например, можно вычислить квадрат этой матрицы:

```
>>> B = A**2
>>> print(B)
[[ 7 10]
 [15 22]]
```

Заметим, что для новой матрицы B нет необходимости явно вызывать конструктор – он будет вызыван автоматически.

Аналогичным образом можно перемножать, складывать и вычитать матрицы, а также умножать матрицы на числа и т.п. (все эти операции уже перегружены в классе `matrix`).

Для более сложных операций над матрицами, например, вычисления обратной матрицы, необходимо импортировать модуль `linalg` пакета `numpy`, либо отдельные функции из него. Например, для той же матрицы A получим:

```
>>> from numpy.linalg import inv
>>> print(inv(A)) # вычисление обратной матрицы
[[-2.   1. ]
 [ 1.5 -0.5]]
```

Matplotlib. Matplotlib – это библиотека двумерной графики для Python, которая позволяет создавать высококачественные рисунки различных форматов. В последних версиях включена также поддержка трехмерной графики. Библиотека позволяет выводить на экран различные графические примитивы, такие как графики функций, гистограммы и т.п. По сути это объектно-ориентированное API над графическими библиотеками wxPython, Qt и GTK+.

Рассматриваемые ниже библиотеки для работы с графами используют внутри себя функции из Matplotlib для формирования изображений графов. Явно нам понадобится вызывать только функцию

`show()` этой библиотеки, если мы хотим вывести рисунок на экран, либо функцию `savefig()` для сохранения рисунка в файл.

NetworkX. Библиотека NetworkX представляет собой мощное средство для работы с различными сетевыми структурами (графами, диаграммами и т.п.), включая их создание, изучение структуры и визуализация. Веб-сайт проекта: https://networkx.github.io/

Следующая программа демонстрирует, как с помощью NetworkX построить граф и вывести его на экран.

```
import networkx as nx
import matplotlib.pyplot as plt
nodes = ['A','B','C','D','E']  # список вершин
edges = [('A','B'), ('A','C'), ('D','B'), ('E','C'),
     ('E','D'), ('B','E'), ('C','D')] # список ребер
G = nx.Graph()           # создаем пустой граф
G.add_nodes_from(nodes)  # добавляем в него вершины
G.add_edges_from(edges)  # добавляем ребра
nx.draw(G, with_labels = True, node_color = 'b')
plt.show()              # вывод изображения на экран
```

В результате будет посроен граф, изображенный на рис. 1 (координаты вершин определяются случайным образом и могут отличаться от тех, что на рисунке).

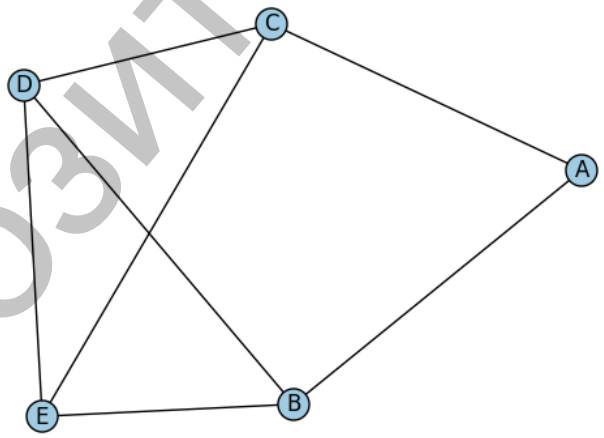


Рис. 1. Построение графа с помощью NetworkX

При желании изображение графа можно сохранить в файл:

```
plt.savefig("graph.png")
```

Цвет вершин можно задать с помощью параметра `node_color` метода `draw()`. В приведенном выше примере был задан общий цвет для всех вершин графа (в данном случае синий – `'b'`). Однако есть

возможность задать индивидуальные цвета для каждой вершины. Для этого нужно параметру `node_color` присвоить список цветов, например,

```
node_color = ['r','g','b','w','y']
```

Класс `Graph` пакета NetworkX определяет неориентированный граф. Если требуется построить орграф, то нужно использовать класс `DiGraph`, точнее вызвать его конструктор:

```
G = nx.DiGraph()
```

Остальной код программы не требует изменения.

Рассмотрим следующую часто встречающуюся задачу: требуется построить граф по заданной матрице смежности. Для этих целей в NetworkX имеется функция `from_numpy_matrix()`. Ниже приведен код программы, которая строит неориентированный граф по заданной матрице смежности.

```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
A = np.matrix([
     [0, 1, 1, 0, 1],
     [1, 0, 1, 0, 0],
     [1, 1, 0, 1, 1],
     [0, 0, 1, 0, 1],
     [1, 0, 1, 1, 0] ])
G = nx.from_numpy_matrix(A)
nx.draw(G, with_labels = True)
plt.show()
```

Аналогично строится ориентированный граф. Единственное отличие в следующей строчке:

```
G = nx.from_numpy_matrix(A, create_using =
     nx.MultiDiGraph())
```

Igraph. Библиотека Igraph – это еще один пакет модулей, предназначенный для работы с графами и сетями. Реализации этой библиотеки доступна на разных языках, включая Python, R и C/C++. Библиотека еще находится в стадии разработки. Она не входит в стандартные репозитории, и поэтому требует ручной установки. Веб-сайт проекта: http://igraph.org.

По своим возможностям библиотека Igraph мало чем отличается от библиотеки NetworkX. Мы не будем останавливаться на ней подробно.

Приведем лишь небольшой пример, демонстрирующий построения графа с помощью Igraph:

```
from igraph import *
G = Graph()       # создаем граф
G.add_vertices(3) # добавляем три вершины
G.add_edges([(0, 1), (1, 2), (2, 0)]) # добавляем ребра
G.vs["label"] = ["A","B","C"]
# присваиваем метки вершинам
plot(G)           # выводим граф на экран
```

Graph-tool. Библиотека Graph-tool представляет собой пакет модулей для статического анализа графов. Сама библиотека написана на языке C++, что обеспечивает высокую производительность. Веб-сайт проекта: https://graph-tool.skewed.de/

В отличие от предыдущих двух библиотек здесь уже реализованы некоторые базовые алгоритмы на графах, например алгоритмы поиска и алгоритмы нахождения максимального потока. Однако мы не будем разбирать эти возможности библиотеки, поскольку в рамках нашего курса эти алгоритмы должны быть реализованы студентами самостоятельно. Поэтому остановимся лишь на базовых возможностях библиотеки Graph-tool.

Рассмотрим пример построения орграфа:

```
from graph_tool.all import *
g = Graph()              # создание объекта типа граф
v0 = g.add_vertex()      # добавление вершин в граф
v1 = g.add_vertex()
v2 = g.add_vertex()
v3 = g.add_vertex()
e1 = g.add_edge(v0, v1) # добавление ребер в граф
e2 = g.add_edge(v1, v2)
e3 = g.add_edge(v1, v3)
e4 = g.add_edge(v2, v0)
e5 = g.add_edge(v2, v3)
graph_draw(g,      vertex_text      =      g.vertex_index,
vertex_font_size = 18, output_size = (200, 200))
```

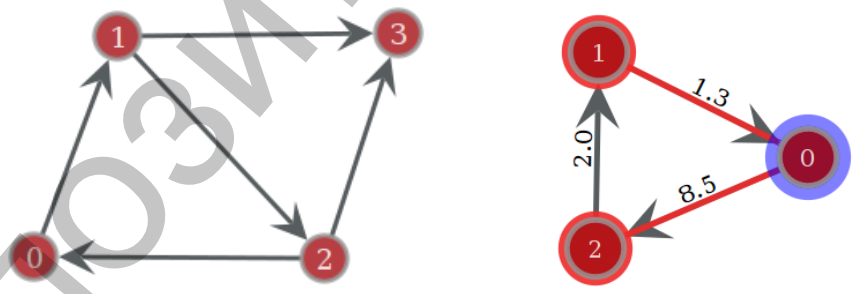
Результат работы программы представлен на рис. 2, *а*.

По умолчанию конструктор класса создает ориентированный граф. Для создания неориентированного графа необходимо присвоить значение `False` атрибуту `directed`:

```
g = Graph(directed = False)
```

Библиотека Graph-tool предоставляет также удобный инструмент для работы с нагруженными графами. Следует использовать метод new_edge_property() для того, чтобы приписать ребрам некоторое свойство, например вес. Рассмотрим пример. Следующая программа строит нагруженный орграф с матрицей весов W и отображает значение веса возле дуг графа (оптия edge_text). Результат показан на рис. 2, *б*.

```
import numpy as np
from graph_tool.all import *
W = np.matrix([
     [0.0, 0.0, 8.5],
     [1.3, 0.0, 0.0],
     [0.0, 2.0, 0.0]])
num_vertices = W.shape[0]   # количество вершин
g = Graph(directed = True)
g.add_vertex(num_vertices)
label_weight = g.new_edge_property('string')
for i in range(0, num_vertices):
     for j in range(0, num_vertices):
          if W[i,j] != 0:
               e = g.add_edge(i, j)     # добавление ребер
               label_weight[e] = str(W[i,j]) # метки ребер
graph_draw(g, vertex_text = g.vertex_index, edge_text =
     label_weight, vertex_font_size = 20,
     edge_font_size = 20, output_size = (300, 300))
```



а ***б***

Рис. 2. Построение орграфов с помощью Graph-tool

На этом мы завершаем обзор библиотек для работы с графами в Python. Для получения более детальной информации используйте документацию к этим библиотекам.

Учебное издание

КУХАРЕВ Андрей Валерьевич
ВИТЬКО Елена Анатольевна
ЦАРЕВ Александр Александрович

АЛГОРИТМЫ НА ГРАФАХ

Методические указания
к выполнению лабораторных работ по дисциплине «Теория графов»

Технический редактор *Г.В. Разбоева*
Компьютерный дизайн *Л.Р. Жигунова*

Подписано в печать 2016. Формат 60x84 $^{1}/_{16}$. Бумага офсетная.
Усл. печ. л. 2,21. Уч.-изд. л. 1,22. Тираж экз. Заказ .

Издатель и полиграфическое исполнение – учреждение образования
«Витебский государственный университет имени П.М. Машерова».

Свидетельство о государственной регистрации в качестве издателя,
изготовителя, распространителя печатных изданий
№ 1/255 от 31.03.2014 г.

Отпечатано на ризографе учреждения образования
«Витебский государственный университет имени П.М. Машерова».
210038, г. Витебск, Московский проспект, 33.