INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Univerzitní centrum podpory

pro studenty se specifickými vzdělávacími potřebami

CZ.1.07/2.2.00/29.0023

Datové struktury a algoritmy samokontroly v Pythonu

KI/DEP

Jiří Fišer, Viktor Mashkov, Lukáš Michalec

Ústí nad Labem 2014

Obor: Informační systémy, Matematická informatika

Klíčová slova: dependabilita, samokontrola, samodiagnostika, Python

Anotace: Tato opora by měla sloužit jako cvičebnice základních datových struktur a algoritmů v oblasti samokontroly s důrazem na přehlednou a v praxi použitelnou implementaci těchto struktur v jazyce Python.
Výhodou použití Pythonu je podpora mechanismů, které implementaci výrazně usnadňují jako jsou OOP třídy a iterátory. Výhodou je i možnost využití standardních vysokoúrovňových knihoven pro numerické a symbolické výpočty a také pro vizualizaci dat.

Projekt „Univerzitní centrum podpory pro studenty se specifickými vzdělávacími potřebami“

Registrační číslo projektu: CZ.1.07/2.2.00/29.0023

Tento projekt byl podpořen z Evropského sociálního fondu a státního rozpočtu České republiky.

Autoři: **Mgr. Jiří Fišer, Ph.D., doc. RNDr. Viktor Mashkov, DrSc., Bc. Lukáš Michalec**

Obsah

1. Příprava

- zjistíte co (a jak) si máte nainstalovat
- poznáte základy Pythonu
- detailněji se seznámíte s pythonskými iterátory

Python, iterátor

1.1 Co potřebuje nainstalovat?

Všechny programové modely v této cvičebnici jsou napsány v jazyce Python a to pro jeho verzi 3.4. Proto je nutné, abyste měli Python v této verzi nainstalován. Nejjednodušší je stažení Pythonu z jeho hlavního distribučního místa (stačí stáhnout a nainstalovat verzi 3.4 nebo novější):

https://www.python.org/downloads/

Pokud máte novější distribuci Linuxu (např. Ubuntu 14.04 a novější), pak máte verzi 3.4 nebo novější nainstalovanou již po běžné instalaci systému (pokud ne, pak stačí instalovat balík *python-3*)

I když je standardní pythonská knihovna velmi rozsáhlá (v souladu s pythonským mottem „baterry included“), ne vždy obsahuje vše co potřebujete. V případě této cvičebnice jsou použity následující doplňkové knihovny:

NumPy — vícerozměrná pole a další rozšíření orientovaná na numerickou matematiku

SciPy — rozšíření knihovny NumPy o moduly vhodné pro řešení vědecko-technických výpočtů

SymPy — symbolické výpočty

Pokud chcete používat i graficky orientované výstupy, pak musíte doinstalovat ještě knihovny:

matplotlib — kreslení různých typů grafů

pillow — manipulace s bitmapovými obrázky (nástupce knihovny PIL)

Tyto knihovny patří k běžně používaným Pythonským knihovnám a lze je nalézt v repozitáři PyPI (*Python Package Index*). Instalace je více než snadná, stačí použít vestavěný program *pip* (může být označen i jako *pip3* nebo *pip3.X*, pokud máte v systému více instalací Pythonu). Stačí na příkazovém řádku uvést příkaz tvaru (jméno balíku vždy obsahuje jen malá písmena):

```
#pip install jméno_balíku
# například
pip install numpy
```

1.2 Co potřebujete znát?

Pro využití této cvičebnici potřebujete tyto předběžné znalosti:

1. **procedurální programování** (podmínky, cykly, funkce) v libovolném procedurálním jazyce (C, Pascal, Java)
2. **základy OOP** (třídy, metody, dědičnost, polymorfismus) v libovolném jazyce s klasickým OOP modelem (Java, C++, PHP)

Programy jsou napsány tak, aby byly srozumitelné i programátorům, kteří využívají jiné programovací jazyky. Proto stačí uvést jen základní přehled. Určitou výjimkou je relativně a časté používání *speciálních metod* a především *iterátorů*. Podobné konstrukce existují i v jiných jazycích, jejich praktické použití je však většinou dosti omezené resp. se liší od přístupu Pythonu. Z tohoto důvodu jsou stručně popsány v dalších dvou podkapitolách. V poslední podkapitole je ještě stručný popis metod pro generování náhodných čísel v Pythonu.

1.3 Přehled syntaxe Pythonu

Základní pravidla syntaxe:

- **Vnořené bloky jsou definovány pouze odsazením (to musí být konzistentní a používají se, pokud možno, jen znak mezery nikoliv tabulátor), Pythonské editory jako je *Komodo Edit* nebo *Eclipse PyDev* vám s odsazováním hodně pomohou.**
- **Příkaz lze napsat jen na jediném logickém řádku. V případě použití závorek však logický řádek končí až uzavřením poslední neuzavřené dvojice závorek (bez ohledu na odřádkování použitá uvnitř). Z tohoto důvodu se někdy v Pythonu používají nadbytečné závorky kolem celých výrazů.**
- **Poznámky jsou pouze jednořádkové, a začínají znakem „#“**

Literály

Literály v Pythonu se výrazně neliší od literálů v ostatních programovacích jazycích.

celočíselné literály: kromě běžných desítkových (12) a šestnáctkových (0x0C) jsou podporovány i oktalové (0o14) a binární (0b1100).

literály s pohyblivou řádovou čárkou: běžný zápis (42.0) včetně semilogaritmického (4.2e1)

boolovské literály: píší se vždy s počáteční verzálkou (*True*, *False*)

Řetězcové literály mohou být v Pythonu uzavřeny jak do uvozovek tak apostrofů. V Pythonu neexistuje znakový typ (znak je prostě jednoznakový řetězec).

Kromě toho lze používat i různé specializované řetězcové literály: *surové* (nejsou v nich rozpoznávány únikové sekvence), *víceřádkové* (mohou obsahovat odřádkování) a jejich kombinace (a to vždy s apostrofy i uvozovkami).

```
"běžný␣literál"
```

```
'běžný␣literál:␣znak␣unicode␣\u2516'

r"surový␣\␣literál"

r'surový␣\␣literál'

"""víceřádkový
literál
"""

r"""
surový␣víceřádkový␣literál,
může␣obsahovat␣přímo␣znaky␣\.
"""

'''
```

```
více řádkový literál
v apostrofech
'''
```

Pokud je literál předcházen znakem *b*, pak je to tzv. *bytový literál* (třída *bytes*), representující buď pole bytových dat (každý znak či úniková sekvence representuje jeden byte) nebo ASCII řetězec (zde je však vhodnější přímo znakový literál). Znaková a bytová data nelze míchat, existuje mezi nimi jen explicitní konverze se specifikací použitého kódování.

```
data=b"header\xA6\02"
data="český text".decode(encoding="iso-latin2")
string = b'Hello, world'.encode(encoding="utf-8")
```

Víceřádkové řetězcové literály se běžně používají jako tzv. *dokumentační řetězce*. Ty následují ihned za hlavičkou třídy, metody nebo funkce a při

běhu programu nehrají žádnou roli (musí však být odsazeny jako tělo dané konstrukce!).

Dokumentační řetězce však lze získat za běhu programu pro účely nápovědy, a dokáží to i statické nástroje, ještě před překladem. Lze tak nabízet nápovědu při zápisu programu v editoru, resp. generovat WWW stránky s dokumentací (např. nástroj *Sphynx* nebo *Dogygen*). Je dobré si na psaní dokumentačních řetězců zvyknout.

Výrazy

Protože Python podporuje valnou většinu operátorů jazyka C (samozřejmě s výjimkou ukazatelových), jsou pythonské výrazy velmi podobné výrazům v Javě, C++, C# a podobných jazycích.

Python však nepodporuje žádné operátory s postranním efektem. Zcela eliminovány jsou operátory inkrementace a dekrementace (++, --). Namísto

operátoru přiřazení je nutno využít přiřazovací příkaz, který se však přiřazovacímu operátoru podobá:

```
i = x + 2 # v Pythonu je to příkaz
```

Dokonce lze psát i kombinovaná přiřazení:

```
i += 2 # zkratka za i = i + 2
i += 1 # nejstručnější opis inkrementace i++
```

Přiřazení však nelze využít uvnitř jiného výrazu:

```
i = j = 0 # nelze
```

Na druhou stranu jsou podporována přiřazení n-tic a obecně sekvencí:

```
i, j = 0, 0
# zkratka za
(i, j) = (0,0)
```

```
# přiřazení položek dvojice
# ale i
i, j, k = [0] * 3 # operátor opakování seznamu
```

Přiřazení položek lze využít i pro výměnu dvou proměnných bez explicitního použití pomocné proměnné (i když ta je skrytě samozřejmě použita)

```
x,y = y,x
```

Specifický tvar má v Pythonu podmínkový výraz (obdoba konstrukce ?: v jazyce C).

```
x if podmínka else y
```

Výraz se vyhodnotí na hodnotu *x*, pokud je podmínka splněna, v opačném případě se vyhodnotí na *y*.

Příkazové konstrukce

V Pythonu existují jen dva druhy cyklů:

Běžný cyklus *while* (s explicitní podmínkou na začátku)

```
while a != 0:
  a, b = b, a % b
```

Mnohem častěji se však využívá cyklus *for* (typově by měl být nazýván spíše *foreach*). Primárně sloužil k procházení seznamů a jiných sekvenčních kontejnerů:

```
for item in seznam:
  print(item)
```

Ale lze jím procházet i posloupnosti čísel (např. indexů).

```
for i in range(10):
  print(seznam[i]) # index
```

V moderním *Pythonu* je cyklus *for* úzce provázán s iterátory (viz níže) a předchozí dvě použití jsou jen speciálními příklady obecnější syntaxe.

Pro vyskakování z cyklů lze použít příkaz *break*. Python podporuje i příkaz *continue* (předčasný přechod k další iteraci). Určitou specialitou Pythonu jsou bloky *else*, které lze použít za cykly. Vykonají se tehdy, je-li cyklus dokončen (tj. skončí s nepravdivou podmínkou nebo dosáhne konce), nikoliv tedy při výskoku nebo výjimce.

Podmíněný příkaz se také neliší od jiných procedurálních jazyků. Protože by však větvení do více než dvou větví vedlo u běžného *if* k vícenásobnému vnoření bloků, je v Pythonu podporována i vícevětvá verze *if-elif-else*:

```
if x < 0:
  return -1
elif x == 0: # zkratka za else if
  return 0
else:
  return 1
```

Funkce

Funkce v Pythonu vždy vrací hodnoty (jsou to tedy skutečné funkce). Procedury (tj. subrutiny bez návratové hodnoty jen s postranním efektem) lze realizovat jako funkce vracející hodnotu *None* (u nich není nutno uvádět příkaz *return*). Protože se neuvádí typ návratové hodnoty (Python je dynamicky typovaný jazyk) nelze funkci od procedury rozlišit podle hlavičky.

Podobně se žádné typové informace neuvádějí ani u formálních parametrů, stačí uvést pouze jména. Na druhou stranu Python podporuje implicitní hodnoty, pojmenované parametry a parametry anonymní (a to jak poziční tak pojmenované). Všechny tyto typy lze různě kombinovat, takže výsledkem může být dosti velký zmatek a to hlavně pro začátečníky. Naštěstí jen u málo funkcí se užívají všechny druhy parametrů.

```
def f ( a, b, c = 3, *, d = 4, *args, **kwargs)
```

Parametry *a*, *b* jsou poziční. Při volání se většinou předávají pozičně, tj. bez uvedení jména, jen musí být uvedeny ve stejném pořadí jako v definici tj. např.

```
f(5,10) # a <— 5, b <—10
```

Lze je však předat i jménem (na pořadí nezáleží, musí se však předat všechny). Pojmenované parametry vždy následují až za pozičními:

```
f(b=10, a=5)
f(5, b=10) # smíšené použití poziční/pojmenované
```

Parametr *c* je poziční, avšak má definovánu implicitní hodnotu. Lze je volat pozičně i jménem, ale nemusí být vůbec předán (pak se použije implicitní hodnota). V Pythonu se implicitní hodnota vyhodnocuje jen jednou při definici funkce, nikoliv při každém jejím volání. To mimo jiné znamená, že by se neměla inicializovat nekonstantními objekty jako jsou seznamy či slovníky!

Poziční parametry lze také předat zabalené do seznamu. Pak se první poziční parametr nastaví podle první položky seznamu, atd.

```
seznam=[5,10]
f(*seznam) # všimněte si jedné hvězdičky
```

Lze použít i inicializaci pomocí slovníku. Pak se klíče slovníku použijí jako

jména parametrů (musí to být řetězce), a příslušné hodnoty se předají.

```
slovnik={"b" : 10, "a" : 5}
f(**slovnik) # u slovníku jsou dvě hvězdičky
```

Formální parametr ***d*** je uveden za položkou obsahující jen hvězdičku. Tyto parametry lze předat **jen jménem** nebo pomocí slovníku, nikoliv pozičně resp. pomocí seznamu.

```
f(5, 10, d="data")
```

Pokud v seznamu formálních parametrů uvedeme parametr začínající hvězdičkou (může mít libovolné jméno, ale konvenčně se užívá téměř výhradně jen ****args***) pak se do tohoto parametru předají všechny přebývající poziční parametry v podobě jejich seznamu (přebytečné parametry lze uvést, jen je-li ****args*** užito). Přebývající poziční parametry mohou být předány i v podobě pole (volání s jednou hvězdičkou).

Podobně se do parametru ****kwargs*** (důležité jsou ty dvě hvězdičky, jméno je však opět konvenční) předají všechny přebývající pojmenované parametry (včetně parametrů předaných slovníkem). K parametru se pak ve funkci přistupuje jako ke slovníku.

```
def g(*args, **kwargs)
```

Pomocí tohoto zápisu lze zpracovat všechny předané parametry. Funkci lze volat s libovolnými pozičními i pojmenovanými parametry (v tomto pořadí!) Zpracování a kontrola počtu parametrů však leží na tíži programátora, nikoliv překladače.

Python podporuje i vytváření bezejmenných (anonymních) funkcí pomocí tzv. **lambda notace** (název pochází z matematické teorie tzv. ***lambda kalkulu*** a je běžně k dispozici ve všech funkcionálních jazycích).

```
lambda x: x + 1 # funkce inkrementace
```

```
# zkratka za (bez zavedení jména)
def inc(x):
  return x + 1
```

Funkce zapsané pomocí lambda notace se běžně využívají jako parametry funkcionálů tj. funkcí, které jsou parametrizované jinými funkcemi. V tomto případě je zbytečné definovat funkci se jménem, které se stejně nikdy nepoužije (je volána jen uvnitř funkcionálu s identifikátorem daného parametru).

```
zip(lambda x,y: (x+y)//2, range(10), range(20,30))
```

Zde je využita anonymní funkce, která průměruje dva své parametry. Ta je následně postupně aplikována na čísla ze dvou rozsahů: 0,1, až 9 a 20,21 až 29.

Kolekce

Python podporuje již v základní syntaxi všechny běžně používané kolekce: n-tice, seznamy, slovníky a množiny. Další lze navíc najít ve standardních knihovnách (hlavní zdroj je modul *collection*).

N-tice (ang. *tuple*) jsou neměnné **sekvence** (= kolekce s indexovatelnou posloupností prvků). Používají se nejčastěji pro krátkodobé seskupení několika objektů, např. pro vracení několika hodnot, resp. při vícenásobném přiřazení. N-tice jsou uzavírány do kulatých závorek (které lze někdy i vynechat)

```
tuple1 = (1,5)
tuple2 = (1,) # jednice
```

Seznam (angl. *list*) je sekvence, do níž lze přidávat prvky a samozřejmě je lze i odebírat. Může být i prázdná, horní mez velikosti je dána jen dostupnou

pamětí. Literál seznamu uzavírá prvky do hranatých závorek:

```
emptyList = []
longList = [1, 2, 3, 4, "pět"]
shortList = list(tuple1)
```

Seznam může být v Pythonu i nehomogenní (tj. tvořený objekty různých nepříbuzných tříd). Nehomogenní seznamy se však obtížně zpracovávají.

Indexace u sekvencí (tj. mimo jiné n-tic a seznamů, ale i řetězců) se neliší od C-like jazyků (index začíná od nuly). Typickým rysem Pythonu je využití tzv. řezů (angl. *slice*). Ty umožňují získávat podsekvence. Zápis `s[d:h:k]` vrací podsekvenci začínající indexem *d* (včetně), obsahující každou k-tou položku a končící před indexem *h* (položka s indexem *h* již není obsažena!). Všechny položky lze vynechat. Dolní mez je implicitně 0 (od začátku), horní velikost sekvence (do konce) a krok je 1 (každý prvek).

Otázka: Jakou funkci má řez s[::-1]?

Slovník (angl. *dictionary*) representuje zobrazení hodnot z množiny klíčů na množinu hodnot. Klíčem mohou být pouze neměnné objekty, které pokud možno definují hashovací funkci (pomocí speciální metody *__hash__*). Běžně se jako klíče používají řetězce, čísla nebo n-tice. Slovník je interně representován jako hashovací tabulka (tj. vyhledávání podle klíče má průměrně konstantní časovou složitost).

```
emptyDict = {}
shortDict = {"jedna" : 1, "dve" : 2}
shortDict2 = {1 : "jedna", 2: "dve"}
# alternativní konstrukce, jsou—li klíči řetězce—identifikátory
ids = dict(jedna=1, dve=2)
```

Množina (angl. *set*) je neuspořádaná kolekce prvků. Hlavní operací je vy-

hledávání (rychlé, interně je to hashovací tabulka) a procházení. Do množiny lze prvky vkládat i vyjímat. Každý prvek může být v množině jen jednou (při pokusu o vícenásobné vkládání se nic neděje). Podporovány jsou i běžné množinové operace (průnik, sjednocení, podmnožina).

```
emptySet = set()
set1 = {"jablko", "hruška"}
set2 = set(longList) # množina ze seznamu
# ztrácí se uspořádání
# a jsou eliminovány vícenásobné výskyty prvků
```

Třída

Python je primárně objektově orientovaný jazyk. Jednou z hlavních konstrukcí jazyka je proto definice třídy. Definice třídy sdružuje v Pythonu pouze definici statických datových členů a instančních metod. Ostatní kon-

strukce (konstruktory, statické metody, vlastnosti) jsou běžné metody se speciální sémantikou (často vytvořené pomocí tzv. dekorátoru). Instanční data se definují v konstruktoru.

Podobně i metody jsou v Pythonu v zásadě jen funkce, které se jen volají speciálním zápisem (`objekt.metoda(parametry)`**) a předávají svého adresáta (objekt před tečkou) jako první parametr. Tento parametr není dokonce ani skrytý, tj. musí být uváděn jako první a má vždy jméno *self* (je to jen úzus, ale velmi silný úzus). Při volání metody je vždy nutno uvést objekt adresáta, tj. i když se volá metoda stejného objektu (je nutno psát** `self.metoda(parametry)`**)**

Python také nepoužívá explicitní specifikace viditelnosti metod resp. datových členů. Vychází se z předpokladu, že veřejné metody jsou ty, které jsou zdokumentovány a označeny jako součást rozhraní. Použití nezdokumentovaných (pomocných) metod je pouze na vlastní bezpečí. Specifikace

v ostatních jazycích stejně mají spíše jen dokumentační charakter a dají se obejít. Pro snadnější orientaci však někteří pythonští programátoři označují neveřejné metody jedním podtržítkem na začátku (tento úzus se používá i zde).

```
class A:
  x = 1 # statický (třídní) datový člen
  def __init__(self, x):
    self.x = x # definice datových členů
    ... # konstruktor
  def m(self, p1, p2):
    ... # instanční metoda
  @property # dekorátor
  def p(self):
    return ... # read—only vlastnost
  @staticmethod
```

```
    def sm(): # není zde self!
      ... # statická (třídní metoda)

objekt = A() # konstrukce objektu
A.x # přístup ke statickému datovému členu
objekt.m(1,2) # volání instanční metody
objekt.p # čtení vlastnosti (bez prázdných závorek!)
A.sm() # volání statické metody
```

Podobně se nesetkáte se žádnými specifikacemi souvisejícími dědičnost metod (*abstract*, *final*, *override* a podobně) nebo rozhraními. Předefinovat lze jakoukoliv metodu a pro dosažení polymorfismu stačí, když dvě třídy sdílejí metodu se stejným jménem a kontraktem (součástí kontraktu často bývá počet a význam jednotlivých parametrů). Nemusí ji dokonce dědit ze stejné třídy (dědičnost je primárně určena pro budování hierarchií tříd

nabízející opakovanou použitelnost děděných metod).

```
class B(A): # dědí z A
  def m(self, p1, p2):
    ... # předefinování metody

objekt = B()
objekt.m(1,2) # volá se předefinovaná verze
objekt.p # volá se zděděná vlastnost
```

Pythonský přístup k polymorfismu je typický pro dynamicky typované jazyky a často se nazývá *duck typing* (kachní typování). Kváká-li něco jako kachna, chodí jako kachna a plave jako kachna, pak je to kachna. Nemusí být opatřena visačkou, co je zač. I v Pythonu je objekt souborem právě tehdy, když umožňuje čtení a zápis a další operace s relativně vágně určeným kontraktem (nemusí být instancí nějaké abstraktní třídy *File*).

V moderní Pythonu se však objevily tzv. abstraktní bázové třídy (*abstract base class*, *ABC*), které slouží jako identifikátory (značky) běžných protokolů. I když nemusí být předky konkrétních tříd, při testování, zda je objekt konkrétní třídy jejich instancí vrací hodnotu *True*.

```
isinstance(2.0, numbers.Integral)
# vrací True pro libovolné číslo nabízející operace
# celých čísel (např. +, —, *, abs, apod.)
isinstance([], colections.abc.Sequence)
# kolekce je sekvencí, tj. má (mimo jiné) metody
# __contains__, __iter__, __reversed__, index, and count
```

Python podporuje i vícenásobnou dědičnost, ale v praxi se příliš nevyužívá (není použita ani zde).

1.4 Speciální metody

Jako speciální se v Pythonu označují metody, které se nevolají standardním zápisem *objekt.metoda()*, ale jsou volány pomocí různých operátorů, vestavěných funkcí resp. jako důsledek různých událostí v životním cyklu objektu.

Speciální metody mají v Pythonu i speciální název: jejich identifikátor začíná dvěma podtržítky a dvěma podtržítky i končí. Tato podtržítka mají zabránit kolizím s identifikátory běžných uživatelsky definovaných metod (v Pythonu by se neměly používat identifikátory se dvěma a více podtržítky na začátku a na konci).

Nejčastěji používanou speciální metodou je konstruktor, jenž se volá po vytvoření prázdného objektu a jehož účelem je vytvoření nového objektu kompozicí z objektů jednodušších. Konstruktor (může být jen jeden) má identifikátor *__init__*. První parametrem je nově vytvořený (prozatím) prázdný

objekt. Tento parametr se vždy označuje jménem *self*.

Další skupinu tvoří speciální metody binárních operátorů. Pokud je například vyhodnocován výraz *a* + *b*, je volán speciální metoda *__add__* nad prvním operandem (tj. zde *a*). Operand *a* se tudíž předá jako první parametr (adresát metody) s konvenčním jménem *self*. Druhý operand (*b*) se předá jako druhý parametr (ten již žádné standardní jméno nemá, volba závisí na kontextu).

```
def __add__(self, right_operand):
  ...
```

Existují i tzv. reverzní metody operátorů začínající písmenem *r* (samozřejmě až po dvojpodtržítku). Zápis a * b vede kromě volání metody *__mul__* nad *a* i k volání metody *__rmul__* nad *b*. V praxi je definována jen jedna z těchto metod.

Pomocí speciálních metod lze dosáhnout i splnění určitého standardního (pod)rozhraní. Objekt tak lze používat v kontextu, jež toto rozhraní používá a v praxi není odlišitelný od vestavěných objektů jež tato rozhraní splňují automaticky.

A) identita objektu je určena obsahem (hodnotová rovnost)

stačí předefinovat operátor rovnosti (==) pomocí speciální metody *__eq__*. Je vhodné předefinovat i metodu *__ne__* (volána na místě operátoru nerovnosti !=).

B) objekty třídy jsou uspořádatelné (porovnatelné vzhledem k operacím menší nebo větší než)

je nutno předefinovat i metody __lt__ (<), __le__ (<=), __gt__ (>), __ge__ (>=).

C) objekt je nemodifikovatelnou sekvencí:

je vhodné předefinovat metody __*len*__ (volána, je-li zjišťována délka sekvence pomocí standardní funkce *len*), a operátor indexace (jen pro čtení) pomocí metody *__getitem__(self, index)*, získání iterátoru pomocí funkce *iter* (speciální metoda se jmenuje *__iter__*) a testovaní, zda je objekt v kolekci (volání operátoru *in* se překládá na metodu *__contains__*).

Speciální metody se používají i pro dvě klíčová přetypování tj. převod na objekt základního typu. Při převodu na řetězec se používá metoda *__str__* resp. *__repr__*. První je volána při explicitním přetypování pomocí funkce *str* resp. při přetypování implicitním. Měla by vracet formát vhodný pro lidského čtenáře (je využívána např. i při debugování). Metoda *__repr__* vrací representaci vhodnou při strojovém zpracování, přičemž tato representace by měla být převoditelná zpět na původní objekt.

Pro převod na logickou hodnotu (užívanou je-li objekt použit jako podmínka v konstrukci *if*) se volá speciální metoda *__bool__*. Měla by vracet hodnotu

True nebo *False*.

Otázka: Kdy je volána speciální metoda *__getattr__*? K čemu ji lze použít?

1.5 Iterátory

Iterátory jsou v Pythonu objekty, které na požádání postupně vracejí objekty s jisté posloupnosti. Nejjednodušší jsou iterátory (potenciálně) nekonečné, které se nikdy nevyčerpají. Běžnější jsou však iterátory konečné, které se po po jisté době vyčerpají a místo dalšího prvku posloupnosti vyvolají výjimku *StopIteration*.

Pro většinu iterátorů je typické tzv. lenivé vyhodnocení, v němž se další hodnota získá až v okamžiku kdy je skutečně potřeba (tj. až když je vyžadován další prvek posloupnosti). U nekonečných iterátorů je toto chování nezbytné (získání všech prvků nekonečného iterátoru předem by vedlo k nekonečně dlouhému provádění), u konečných může výrazně zvýšit paměťovou efektivitu jejich implementace (není potřeba uchovávat či kopírovat

všechny prvky)

Získání iterátoru

Iterátory lze nejjednodušeji získat aplikací vestavěné funkce *iter* na tzv. iterovatelné objekty (= objekty se speciální metodou *__iter__*).

Nejčastěji se používá iterátor, jenž postupně vrací celá čísla v určitém rozmezí (a s určitým krokem). Ten lze získat aplikací funkce *iter* na objekt vracený vestavěnou funkcí *range*. Tato funkce má obecně tři parametry *min*, *max* a *krok*. Získaný iterátor vrací postupně čísla min, $min + krok$, $min + 2 \times krok$, atd. až do horní meze (vyjma, tj. horní mez je z iterátoru vždy vyloučena). Některé z parametrů funkce lze vynechat, pak se použijí rozumné implicitní hodnoty podobně jak u řezů (rozumné hlavně z hlediska indexace, která v Pythonu začíná od nuly).

```
iter(range(0,8,2)) -> iterátor poskytující hodnoty 0,2,4,6
```

```
iter(range(0,8)) -> iterátor vracející hodnoty 0,1,2,3,4,5,6,
iter(range(8)) -> iterátor vracející hodnoty 0,1,2,3,4,5,6,7
```

Iterovatelný je i objekt textového proudu získaný funkcí *open*. Iterátor, jenž lze i v tomto případě získat voláním funkce *iter*, postupně vrací jednotlivé řádky souboru. Iterátor je samozřejmě lenivý, tj. přečten je jen aktuálně požadovaný řádek.

Nejběžnějšími iterovatelnými objekty jsou však sekvenční kolekce. Iterátor, který získáme aplikací funkce *iter* na sekvenci (seznam, n-tici, množinu, apod.), vrací postupně jednotlivé prvky kolekce (u seznamů a n-tic v pořadí od prvního po poslední).

Slovníkové kolekce nejsou přímo iterovatelné, ale poskytují metody, které vrací iterovatelné pohledy na slovník. Metoda *items()* vrací iterovatelný objekt vracející postupně dvojice *(klíč, hodnota)*, metoda *keys()* umožňuje pohodlné postupné procházení klíčů.

Využití iterátorů

Pro získání dalšího prvku z iterátoru stačí zavolat vestavěnou funkce *next*. Explicitní volání funkce *next* se však používá jen velmi zřídka, neboť procházení iterátorů výrazně usnadňuje cyklus *for(each)* a několik tzv. komprehenzí (zkrácených zápisu cyklu).

V obou případech není argumentem přímo iterátor, ale iterovatelný objekt. Na ten je použita funkce *iter*, která vrací skutečně použitý iterátor. Pro zjednodušení platí, že i iterátor je iterovatelným objektem (pokud na něj aplikujeme funkci *iter*, je vrácen původní iterátor).

Cyklus *for* postupně získává prvky z iterátoru a přiřazuje odkaz na ně do lokální proměnné. Nad jednotlivými prvky lze pak provádět libovolné operace (prvek je v těle cyklu odkazován proměnnou). Po dosažení konce iterátoru cyklus samozřejmě skončí.

```
for i in range(10): # argumentem je iterovatelný objekt
  print(i)
for person in list: # argumentem je iterovatelný objekt
  print(person)
for person in iter(list): # argumentem je přímo iterátor
```

Iterátorová komprehenze se trochu podobá cyklu *for*, je však na rozdíl od něj výrazem tj. po vyhodnocení vrací hodnotu. Touto hodnotou je nový iterátor, který vznikne modifikací původního iterátoru. Při modifikaci mohou být transformovány jednotlivé prvky a některé mohou být z posloupnosti vyňaty.

```
(x*x for x in range(1,10))
```

Výsledkem tohoto zápisu (závorky kolem jsou zde povinné!) je iterátor vracející čísla 1, 4, 9, 16 až 81, tj. druhé mocniny prvních devíti přirozených čísel.

Podobně:

```
sum(x*x for x in range(1,10))
```

vrací součet těchto druhých mocnin tj.

$$\sum_{i=1}^{9} i^2$$

Všimněte si, že kolem komprehenze nemusí být zvláštní závorky, pokud je tato jediným parametrem funkce (stačí závorky používané při volání funkce).

Typický pythonský idiom — spojení řetězcové representace položek seznamu do jediného řetězce za použití čárky jako oddělovače:

```
",".join(str(item) for item in seznam)
```

Otázka: Jaká je výhoda spojování řetězců pomocí metody *join* oproti spojování řetězců operátorem plus? (nápověda: řetězec je i v Pythonu neměnný).

Testování zda jsou všechna čísla v seznamu kladná (jen v tomto případě vrací *True*):

```
all(i>0 for i in seznam)
```

Kromě transformace hodnot iterátoru lze některé položky z iterátoru vyjmout:

```
(i for i in seznam if 10 <= i <= 20)
```

Nový iterátor poskytuje jen ty prvky původního iterátoru, které leží mezi 10 a 20 včetně.

Zobrazení (mapování) a filtrování hodnot lze samozřejmě kombinovat.

```
((s if len(s) <= 10 else s[0:10])
 for s in seznam if s is not None)
```

vrací iterátor poskytující řetězce z původního seznamu, přičemž odstraňuje hodnoty *None* (*None* je podobné *null* známé z jiných jazyků, vyjadřuje nepřítomnost hodnoty) a dlouhé řetězce zkracuje na 10 znaků.

Kromě iterátorových komprehenzí existují i komprehenze, které namísto iterátorů vrací kolekce (seznamy, množiny či slovníky). Tím se však ztrácí výhoda lenivého vyhodnocení, neboť iterátor musí být spotřebován celý (tj. musí být vyčísleny všechny jeho položky). Na druhou stranu lze kolekci procházet i vícekrát (iterátory lze využít vždy jen jednou). Seznamy lze navíc procházet i v opačném pořadí. Následující příkaz vrací seznam čísel 1 až 1000 (včetně). Tento seznam zaujme v paměti několik KiB.

```
[i for i in range(1,1001)]
# lze zapsat i pomocí konstruktoru seznamu
# a iterátorové komprehenze
list(i for i in range(1,1001))
```

Komprehenze lze samozřejmě vnořovat (vytváří jednotkovou matici representovanou jako seznam seznamů) :

```
[[(1 if x == y else 0) for x in range(10)]
 for y in range(10)]
```

Podobně lze konstruovat i množiny (zde je vytvářena tj. množina dvojic z kartézského součinu vyjímaje dvojice stejných prvků):

```
{(x,y) for x in [1,2] for y in [1,2,3] if x != y}
# vrací {(1, 2), (1, 3), (2, 3), (2, 1)}
```

Jen o mírně složitější je slovníková komprehenze (slovník mapující tři reálná čísla na jejich logaritmy):

```
{x:math.log(x) for x in [1,0, 2.0, 3.0]}
```

Kombinování iterátorů

Složitější iterátory lze vytvářet i kombinací již existujících.

Nejjednodušším případem je kombinace iterátoru s posloupností celých čísel, které poté slouží jako indikátor pořadí (první pozice je však v souladu s principy Pythonu „indexována“ číslem 0).

Kombinaci provádí vestavěná funkce *enumerate*. Přijímá původní iterátor a vrací iterátor poskytující uspořádané dvojice (0, p_0), (1, p_1)...(1, p_n), kde p_0, p_1,..., p_n jsou prvky původního iterátoru.

Funkce *enumerate* se běžně používá v cyklu *for*:

```
for i, item for enumerate(seznam):
   print("{0}. polozka = {1}".format(i, item))
```

N-tice poskytované rozšířeným iterátorem jsou přiřazeny do dvojice proměnnách *i*, *item*, přičemž do *i* je vloženo pořadí, do *item* prvek původního iterátoru.

Toto spojení dvou posloupností (proudů) hodnot lze zobecnit pomocí vestavěné funkce *zip*. Tato funkce přijímá n-iterátorů a vrací jeden nový, který poskytuje n-tice, které jsou vytvořeny z těch prvků původních iterátorů, které byly poskytnuty původními iterátory na stejné pozici. Tj. i-tá n-tice je tvořena i-tými prvky původních iterátorů. Funkce se ukončí v okamžiku, kdy je vyčerpán nejkratší iterátor.

Název funkce pochází z funkční podobnosti s běžným zipem (zdrhovadlem). I zip spojuje dohromady zuby, které jsou na odpovídajících pásech ve stejné

pozici (i když zde jsou zuby vůči sobě mírně posunuty). V matematice se tato funkce označuje jako konvoluce.

Funkci *enumerate* lze implementovat pomocí funkce *zip*. Stačí jen zazipovat nekonečný číselný iterátor s iterátorem, který chceme enumerovat (= opatřit pozičními indexy):

```
import itertools
zip(itertools.count(), seznam)
```

Funkce *count* z modulu *itertools* poskytuje nekonečný iterátor od daného čísla (implicitně 0) s daným krokem (implicitně 1).

Funkce *zip* však nabízí i další možnosti své aplikace:

```
(i for i,r
 in zip(itertools.count(1),
      iter(lambda: random.uniform(0,1), None))
 if r < 1/n)
```

Vyhodnocením tohoto výrazu vzniká nekonečný iterátor, který vrací výběr z celých čísel počínaje 1. Průměrně každé n-té číslo (n je zadáno na konci výrazu), vzdálenosti mezi nimi jsou však náhodné. Zipovány jsou dvě sekvence: první je posloupnost čísel od nuly výše (po jedné), druhou je (nekonečná) posloupnost náhodných čísel s rovnoměrným rozdělením nad intervalem [0,1]. Ta je získána speciální formu funkce *iter*, kterou lze volat i nad bezparametrickými funkcemi (zde je to funkce získaná z funkce *random.uniform* fixací jejích dvou parametrů). Iterátor se získá opakovaným voláním funkce, dokud funkce nevrátí hodnotu určenou druhým parametrem. V našem případě je zarážkou hodnota *None*, kterou však funkce

random.uniform **nikdy nevrátí (vrací vždy jen čísla) a iterátor je tak nekonečný. Zip tyto iterátory spojí do jednoho iterátoru vracející dvojice. Mohou to být např. dvojice (1, 0.58669), (2, 0.02566), (3, 0.856711) atd. Iterátorová komprehenze pak vybere jen ty dvojice, jejichž druhý člen je menší než *1/n* (tj. průměrně každé n-té číslo) a vrátí iterátor poskytující jen prvý člen dvojice.**

Posloupnost, který je generována tímto zápisem, modeluje ruskou ruletu s n zásobníky (jen v jednom je náboj) a s rotací zásobníku. Pravděpodobnost, že vzdálenost mezi čísly je *n* má binomické rozdělení.

Generátory iterátorů

Generátory iterátorů nabízejí nejobecnější prostředek pro vytváření iterátory. Lze s nimi snadno vytvořit všechny výše uvedené iterátory a to je jen zlomek jejich vyjadřovací síly. Na rozdíl od výše uvedených konstrukcí však

vyžadují vytvoření tzv. korutina což je blízká obdoba běžných funkcí a metod.

Rozdíl je v době života každé instance funkce resp. korutiny. U funkcí (či metod resp. procedur obecně tzv. subrutin) vzniká instance funkce po jejím zavolání. Tato instance má své vlastní lokální proměnné a vlastní instance parametrů, tvořící kontext v němž se vykoná její kód. Po ukončení funkce tato instance a její kontext zaniká a jediný přeživší výsledek existence této instance je případná návratová hodnota. Při novém volání funkce se vytvoří nová instance s novým kontextem, zcela odděleným od instance předchozí. Řečeno podle Hérakleita: dvakrát nevstoupíš do stejné řeky resp. subrutiny (originál je ještě inspirativnější: Ποταμοῖς τοῖς αὐτοῖς ἐμβαίνομέν τε καὶ οὐκ ἐμβαίνομεν **tj. zároveň vstoupíš i nevstoupíš).**

Korutina je je v tomto ohledu mnohem pružnější. Po jejím zavolání a vytvoření kontextu ji lze opustit (s návratovou hodnotou), provést jinou rutinu

a pak se do původní korutiny vrátit (je to stále ta samá instance se všemi původními hodnotami proměnných i parametrů). Po následném běhu programu uvnitř této instance korutiny ji lze znovu přerušit. Počet přerušení není omezen, i když i korutiny nakonec skončí a jejich instance zanikne. Jinak řečeno subrutina má v čase jen jeden vstupní bod a jeden výstupní, zatímco korutina má v čase neomezený počet vstupů i výstupů.

Jak to souvisí s iterátory? Při každém (dočasném) opuštění korutiny je vrácen jeden objekt. Korutinu tak lze využít pro vytvoření iterátoru, který může být nekonečný i konečný (je-li korutina opuštěna definitivně).

```
def count(init):
  i = init
  while True:
    yield i
    i = i + 1
```

Pokud tuto korutinu zavoláme (běžným zápisem např. `count(2)`**), tak získáme objekt, který se chová jako iterátor. Na počátku se žádný kód korutiny neprovede (korutina je hned pozastavena) a řízení je předáno kódu který korutinu zavolal.**

Až v okamžiku, kdy nad iterátorem poprvé zavoláme funkci *next* (typicky v cyklu nebo seznamové komprehenzi), se vykoná začátek kódu korutiny. Je inicializována lokální proměnná a tok řízení vstoupí do nekonečného cyklu. Hned první příkaz v těle cyklu (*yield*) však korutinu pozastaví, řízení se vrátí do metody *next* a ta vrátí hodnotu, která je argumentem příkazu *yield* (v našem případě tedy 2) . Po určité době se program pokusí získat další hodnotu z iterátoru a tak je nad ním opět volána funkce *next*. Ta obnoví běh rutiny v bodě, kde byla opuštěna a pokračuje v provádění těla cyklu. Kontext zůstal zachován, tj. hodnota proměnné *i* zůstala zachována (spolu s hodnotou parametru, ten nás zde příliš nezajímá). Proto je v druhém pří-

kazu původní hodnota zvýšena o jedničku, a cyklus vstoupí do další iterace. Tam je opět spolu s korutinou přerušen a iterátor vrátí aktuální hodnotu proměnné i (tj. v našem případě 3). Protože je cyklus nekonečný, tak lze volat metodu *next* nad iterátorem neomezeně-krát a získávat tak další čísla v posloupnosti.

Je zřejmé, že podobně lze pomocí iterátorového generátoru (= korutiny) vytvořit všechny výše uvedené iterátory. U tak jednoduchých iterátorů je to však zbytečné, neboť použití iterátorových funkcí je jednodušší (i když u ruské rulety si již nejsem tak zcela jist). Generátory se používají především v případě, kdy je iterátor komplikovanější nebo je použit na více místech. Je to podobné jako u výrazů a funkcí.

Velmi užitečné jsou iterátory především v případě, kdy je stav iterátoru složitější než jediná hodnota (tou hodnotou může být např. index do kolekce). Hodí se také pro vytváření různých funkcí nad iterátory (funkce *zip* a funkce

v *itertools* jsou definovány pomocí generátorů, jejich kód je možno nalézt v pythonské dokumentaci).

Nejdříve si ukažme dvě užitečné funkce nad iterátory:

```
def take(n, iterator):
   for i in range(n):
      yield next(iterator)

def step(n, iterator):
   for i in range(n):
      next(iterator)
   while True:
      yield next(iterator)
```

První korutina vrací původní iterátor, který poskytuje stejné objekty jako původní (druhý parametr), avšak maximálně *n* prvků. Je např. vhodný pro

zkrácení nekonečných iterátorů (pokud nás zajímá jen prvních *n* položek). Druhý generátor naopak prvních *n* členů původních iterátorů přeskočí (= zahodí).

Pokud použijeme obě funkce společně není problém udělat libovolný výřez z iterátoru:

```
print(list( take(5, step(10, iter(range(100))))))
```

Výsledný iterátor poskytuje položky s pořadím 11, 12, 13, 14, a 15 z původního číselného iterátoru (přeskočí prvních deset a pak vrátí pět). Iterátor je převeden na seznam a ten je vypsán (vypíše se [10, 11, 12, 13, 14]).

Zajímavější je vytvoření iterátoru, který poskytuje členy známe Fibonacciho posloupnosti:

```
def fibi():
    a = 1
    b = 1
    while True:
        yield a
        a, b = b, a + b
        # přiřadí zároveň a <—b a b <— a + b
```

Iterátor je nekonečný, takže pro výpis je nutné použít např. naši funkci *take*:

```
print(list(take(20, fibi())))
```

Pokud využijeme skutečnosti, že i tento iterátor je lenivý (na požádání vždy vypočte a vrátí jen jeden člen) a že uvnitř generátoru lze procházet další generátory, pak existuje i elegantnější (ale obtížněji pochopitelná) implementace generátoru Fibonacciho posloupnosti:

```
def rfibi():
   yield 1
   yield 1
   for f1, f2 in zip(rfibi(), step(1, rfibi())):
      yield f1 + f2
```

Korutina poskytuje součet prvků dvou (lenivě vyhodnocovaných) posloupností — rekurzívně získané Fibonacciho posloupnosti a Fibonacciho posloupnosti bez prvního prvku (jakoby byla je posunuta o položku vpravo). Je to jakýsi multi-uroboros.

Otázka: Jaká je výhoda této rekurzivní verze výpočtu Fibonacciho posloupnosti oproti klasickému rekurentnímu algoritmu? (rada: ověřte časovou a paměťovou složitost algoritmů)

1.6 Generování náhodných čísel

Při tvorbě simulačních aplikací hrají klíčovou roli generátory náhodných čísel. V reálných systémech probíhají nízkoúrovňové (fyzikální) procesy, které produkují viditelné výstupy (výrobky či informace) a občas bohužel prostřednictvím interních selhání (*faults*) i chyby.

V simulačním programu nejsou tyto procesy simulovány (v zásadě ani být simulovány nemohou, neboť simulace na subatomární úrovni je současnými prostředky nemožná), ale jsou nahrazeny náhodnými výstupy podle určitého statistického rozdělení. Tato rozdělení se získávají statistickým rozborem reálných systémů. Podobně jsou pomocí náhodných veličin simulovány i časové průběhy produkce výstupů resp. vzniku selhání.

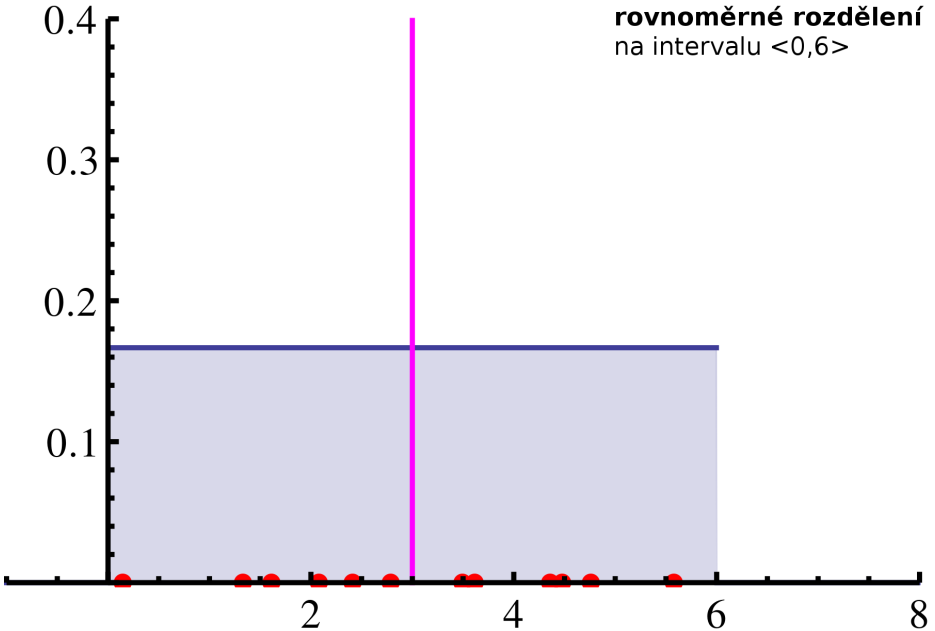
Generátory náhodných čísel tak tvoří klíčovou součást simulačních programů. Na rozdíl od herních aplikací však nepostačuje pouze generátor s rovnoměrným rozdělením (elektronický ekvivalent hrací kostky), ale je potřeba

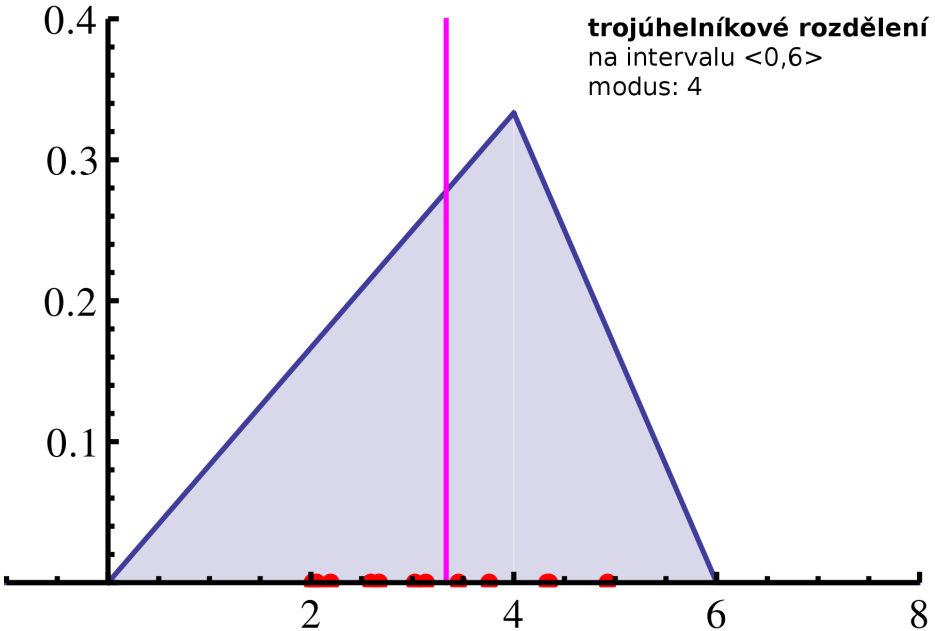
implementovat i generátory s jinými statistickými rozděleními (ty se většinou implementují pomocí filtrů nad původním generátorem s rovnoměrným rozdělením).

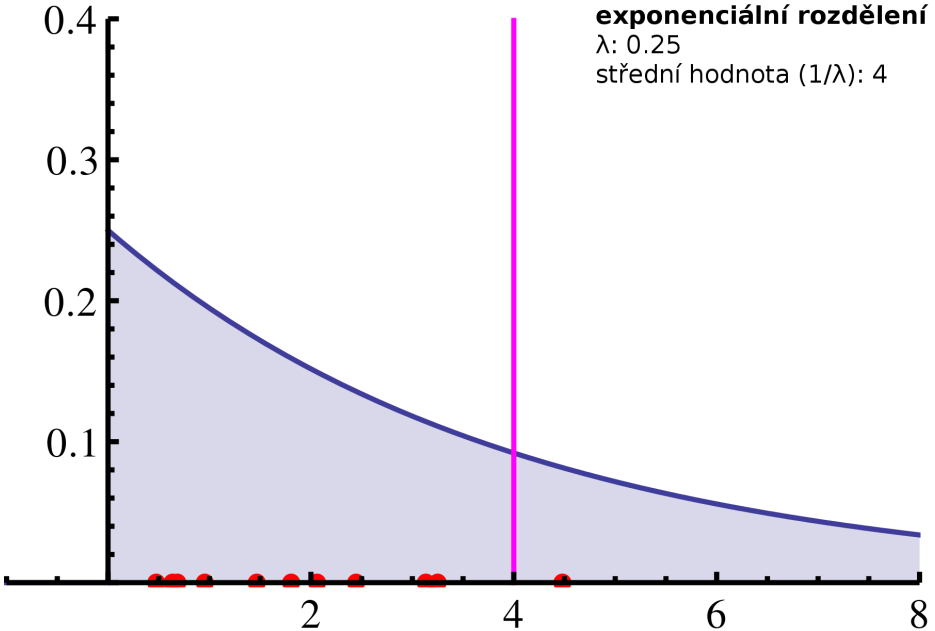
Pythonský modul *random* naštěstí tento nezbytný předpoklad splňuje. Používá kvalitní zdroj náhodnosti (generátor typu *Mersenne Twister* s periodou $2^{19937} - 1$ a 53 bitovou přesností) a podporuje valnou většinu běžně používaných rozdělení.

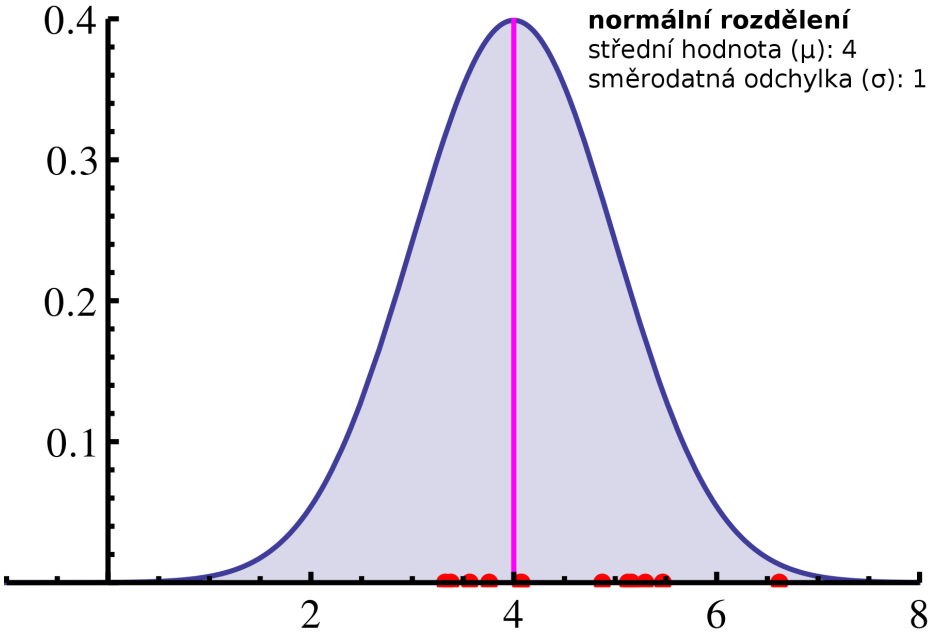
funkce	popis
uniform(a,b)	rovnoměrné rozdělení z interv. [a, b]
triangular(a,b,mod)	trojúhelníkové z intervalu [a,b] s daným modem (maxim. výskytů)
expovariate(λ)	exponeciální se střední hodnotou 1/λ
normvariate(μ, σ)	normální (μ = střední hodnota, σ = standardní odchylka)
choice(seq)	vrací jeden náhodný prvek ze sekvence
sample(seq, k)	vrací *k*-prvkový náhodný výběr unikátních prvků ze sekvence

Pro jistotu připojuji grafy rozdělení pravděpodobnosti pro všechna čtyři výše uvedená rozdělení. Červené body na ose *x* ukazují deset náhodně vygenerovaných hodnot s daným rozdělením, svislá purpurová čára pak střední hodnotu daného rozdělení.


rovnoměrné rozdělení
na intervalu <0,6>
0.4
0.3
0.2
0.1
2
4
6
8


0.4
0.3
0.2
0.1
2
4
6
8
trojúhelníkové rozdělení
na intervalu <0,6>
modus: 4


exponenciální rozdělení
λ: 0.25
střední hodnota (1/λ): 4
0.4
0.3
0.2
0.1
2
4
6
8


0.4
0.3
0.2
0.1
2
4
6
8
normální rozdělení
střední hodnota (μ): 4
směrodatná odchylka (σ): 1

Grafy nejsou vytvořeny v Pythonu (i když pomocí rozšíření *SciPy* a *Matplotlib* by to bylo možné), ale pomocí programu *Mathematica 8*. Pro vygenerování postačily dva (delší) řádky kódu (pro informaci *PDF* vrací funkci pravděpodobnosti pro dané rozdělení, a operátor /@ mapuje funkci na seznam).

```
dists = {UniformDistribution[{0, 6}], TriangularDistribution[{0, 6}, 4],
       ExponentialDistribution[1/4], NormalDistribution[4, 1]}

Show[Plot[PDF[#, x], {x, -1, 8},
      AxesStyle -> Directive[Thick, 16],
      PlotStyle -> Thick, Filling -> Bottom,
      PlotRange -> {{-1, 8}, {0, 0.4}}
    ],
    Graphics[{Red, PointSize[Large],Point[Table[{RandomVariate[#],0},{12}]],
           Thick, Magenta, Line[{{Mean[#], 0}, {Mean[#], 0.4}}]}
    ]
] & /@ dists
```

2. Diagnostický graf systémů se samokontrolou

- poznáte systémový přístup k samodiagnostice
- naučíte se representovat systémy se samodiagnostikou
- vytvoříte základní implementaci modelu systému se samodiagnostikou

samodiagnostika, diagnostický graf

2.1 Teorie

samodiagnostika — vzájemná kontrola a diagnostika jednotlivých zařízení komplexního stroje. Neexistuje zde žádné dedikované kontrolní zařízení, tj. zařízení vykonávají jak běžnou tak kontrolní činnost.

modul — abstrakce zařízení na systémové úrovni. Modul provádí běžnou

činnost a zároveň i kontrolu ostatních modulů.

atomická kontrola — prováděná modulem a jiném modulu, v praxi o bývá např. jednoduché porovnání výstupních dat ze zařízení s daty, která jsou považována za správná (etalonní), kontrolní součty, apod.

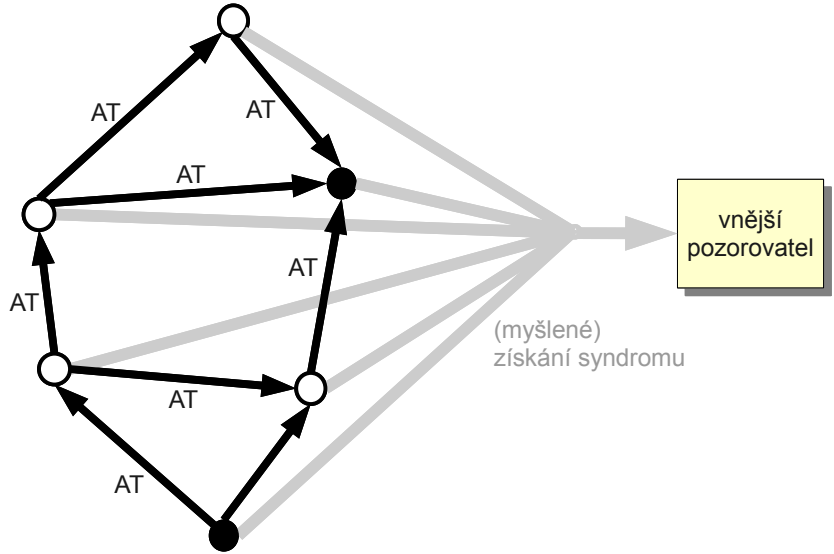
Celkový model systému je representován grafem modulů, kde uzly representují moduly tj. jednotlivá dílčí zařízení a hrany atomické kontroly. Tento graf se nazývá **diagnostický graf** systému (zkráceně DG).

Každé technické zařízení (tj. modul) může být buď ve stavu, kdy poskytuje správná výstupní data (= **bezchybný modul**) resp. ve stavu kdy jsou produkovaná data nesprávná (= modul selhal, **chybný modu**l). Bohužel může (kontrolující) zařízení chybně vyhodnotit data přijatá z druhého zařízení a to v obou směrech (správná označit za chybná resp, chybná za správná) a tak zařízení ohodnotit nesprávně.

Každý modul tak má svá vlastní hodnocení modelů, které zkontroloval a

tato hodnocení nemusí být konzistentní (tj. neexistuje konsensus v hodnocení jednotlivých modulů). Navíc pravděpodobnost nekonzistence může být relativně velká.

Abstraktní vnější pozorovatel, dostává výsledky všech dílčích atomických kontrol (a nedojde přitom k žádnému chybnému přenosu nebo chybné interpretaci obdržených dat. Tento pozorovatel je ve skutečnosti jen myšlený, neboť v praxi musí být kontrola plně autonomní tj . zajištěna pouze moduly systému. Zavedení vnějšího pozorovatele však usnadní počáteční popis systému.



Diagnostika na systémové úrovni spočívá v odhalení všech chybných (=selhávajících) modulů. Naopak na nás na této úrovni nezajímá, co je konkrétní

příčinou selhání modulu (tj. co se stalo uvnitř zařízení).

V našem (zjednodušeném) modulu je jediným vstupem diagnostiky (prováděné abstraktním *vnějším pozorovatelem*) množina výsledků všech provedených atomických kontrol. Tato množina je označována jako syndrom. Výstupem je seznam chybných modulů resp. komplementární seznam modulů bezchybných.

Výstup, tj. diagnostika i její důvěryhodnost, závisí na několika předpokladech souvisejících s atomickými kontrolami. Nejdůležitějším je předpoklad o výsledcích kontrol prováděných jak bezchybnými tak selhávajícími moduly.

Uvažujme například atomickou kontrolu modulu M_j modulem M_i. Lze předpokládat, že výsledek kontroly bezchybného modulu bezchybným modulem je roven vždy hodnotě 0 (notace má tvar $r_{ij} = 0$).

Otázka: Platí tento předpoklad opravdu vždy? Za jakých okolností může být narušen?

Při kontrole selhávajícího modulu (zde tedy např. M_j) modulem bezchybným (zde např. M_i) se většinou předpokládá, že r_{ij} je v tomto případě rovno 1, tj. bezchybný modul vždy odhalí chybu v modulu selhávajícím.

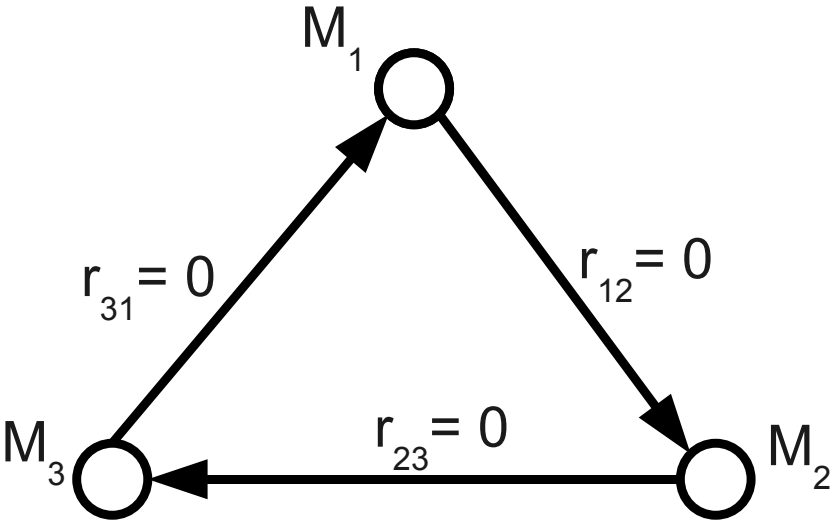
Nejsložitější je situace v případě že kontrolu provádí selhávající modul. V tomto případě však lze předpokládat, že výsledek bude náhodný, tj. může nabývat jak hodnoty 0 tak 1. V nejjednodušším případě budou tyto hodnoty nabývat se stejnou pravděpodobností ($\frac{1}{2}$).

Výsledek atomické kontroly modulu M_j modulem M_i lze tudíž definovat takto:

$$r_{ij} = \begin{cases} 0 & \text{jsou-li oba moduly } M_i \text{ i } M_j \text{ bezchybné} \\ 1 & \text{je-li } M_i \text{ bezchybný a } M_j \text{ selhávající} \\ X(0,1) & \text{je-li } M_i \text{ selhávající} \end{cases} \quad (1)$$

Výsledky atomických kontrol lze representovat pomocí ohodnocení hran a to i grafické podobě (vhodné pro lidské pozorovatele).

Syndrom na následujícím obrázku umožňuje učinit pouhým pohledem závěr, že všechny tři moduly jsou bezchybné (za předpokladu výše uvedeného modelu výsledků atomických kontrol).


M1
r31 = 0
r12 = 0
M3
r23 = 0
M2

Otázka: Jaký by byl závěr, pokud by výsledkem atomické kontroly modulu M_3 modulem M_2 byla hodnota 1 ($r_{23} = 1$).

2.2 Implementace

Pro lepší poznání systémové úrovně samodiagnostických systémů je vhodné vytvořit si jednoduchý programový model, který lze využít pro simulaci reálných systémů. Tento model bude samozřejmě velmi abstraktní, neboť bude zohledňovat jen samodiagnostiku na systémové úrovni. Implementace je tak omezena pouze na operace související se samokontrolou a samodiagnostikou (není modelována běžná činnost modulů), a i ta je výrazně zjednodušena (tj. nemusí být např. modelována interní logika a realizace atomických kontrol.

Při návrhu libovolného systému lze postupovat od návrhu hlavní centrální komponenty po komponenty dílčí (zde od návrhu objektu celého systému

po návrh modulů). Tento návrhový styl se označuje jako návrh odshora dolů. Návrh odspoda nahoru naopak začíná u nejelementárnějších objektů (zde tedy modulů), které skládá do složitější objektů (zde do celkového systému). Každý z těchto přístupů má své výhody a nevýhody, a tak se běžně v komplexnějších návrzích oba přístupy kombinují.

My zde použijeme (o něco méně běžný) postup seshora dolů. Výhodou tohoto přístupu je snadnější dodržení dostatečné úrovně abstrakce na každé z úrovní. Prakticky nevznikají situace, kdy objekt omylem zviditelňuje svou vnitřní implementaci (at totiž není ještě navržena resp. dokonce implementována). Také prakticky nedochází k implementaci nadměrně abstraktních činnosti na zbytečně nízkých úrovních.

Třída System

Začneme tudíž od třídy representující celý systém.

Nejdříve se podívejme co od objektů třídy *System* primárně očekáváme:

1. **schopnost snadno vytvářet systémy modulů včetně jejich diagnostických grafů**
2. **získávat výsledný syndrom aplikací všech atomických kontrol**

Pro účely použití vnějším pomocným kódem (například pro vizualizaci systému) se ještě hodí export dat systému v rozumných formátech. Nejpřenositelnějším je matice sousednosti diagnostického grafu, ta však nenese informaci o modulech (například interní identifikaci či typ modulů). Proto je nutné doplnit minimálně iterátor přes moduly.

Jedním z klíčových designérských rozhodnutí je mechanismus identifikace uzlů. V zásadě existují dvě možnosti:

1. **interní identifikace modulů tj. každý modul si nese vlastní identifikátor (což může být de facto libovolný řetězec či dokonce objekt). Je**

srozumitelnější, pružnější avšak pro některé representace nevhodný (matice sousednosti). Posiluje také samostatnost modulů (lze je zpracovat i mimo systém či sdílet mezi systémy).

2. vnější identifikace pozicí (indexem) v systému. Méně pružné a závislé na vnitřní implementaci (nepříliš vhodné především pro nehomogenní systémy). Na druhou stranu efektivní a pro většinu aplikací dostatečné.

Nakonec jsem se rozhodl implementovat obě možnosti (tj. modul je identifikovatelný jak pomocí svého vlastního identifikátoru tak i indexem). Navenek se tato dichotomie projevuje nutností definovat metody, které zajistí překlad indexu na modul (a tím i jeho interní identifikátor) resp. modulu (s identifikátorem) na index.

Začneme proto programovým (třídním) modelem systému všech modulů.

```
class System:
    '''
        Systém modulů propojených atomickými kontrolami.
    '''
    def __init__(self, modules=None, graph=""):
        '''
            vytváří systém s danými moduly a diagnostickým grafem.
            Pořadí modulů je pro některé operace signifikatní (např.
            při vytváření matice sousednosti).
            :type modules: list
            :type graph: str
        '''
        self.moduleList = list(modules)
        self.modules = {m.id: m for m in self.moduleList}
        self.size = len(self.modules)
```

```
        self.dg = set()
        self._processGraphInput(graph)

    def _processGraphInput(self, graph):
      # parsuje stručné zápisy hran diagn. grafu
        for path in graph.split(";"):
            lastModulSet = set()
            for group in path.split("-"):
                modulSet = {self.modules[ident]
                            for ident in group.split(",")}
                for m_i in lastModulSet:
                    for m_j in modulSet:
                        self.dg.add(AtomicCheck(m_i=m_i, m_j=m_j))
                lastModulSet = modulSet

    def getSyndrome(self):
```

```
        '''
            vrací syndrom získaný jednorázovou aplikací všech
            atomických testů.
        '''
        s = Syndrome(self)
        for (m_i, m_j) in self.dg:
            s.addResult((m_i, m_j), m_i >> m_j)
        return s

    def toAdjacencyMatrix(self):
        '''
            vrací matici sousednosti diagnostického grafu jako
            objekt třídy numpy.matrix.
        '''
        matrix = np.matrix(np.zeros((self.size, self.size),
                                    dtype=np.int8),
```

```
                    copy=False)
        for (m_i, m_j) in self.dg:
            matrix[self.mpos(m_i), self.mpos(m_j)] = 1
        return matrix

    def mpos(self, module):
        '''
            vrací index daného modulu v rámci systému. Index je dán
            pozicí modulu v konstruktoru. Index je v rozsahu [0, n).
            :type module: Module
        '''
        return self.moduleList.index(module)

    def __str__(self): # výpis primárně pro účely ladění
        mods = ", ".join(str(m) for m in self.moduleList)
        tests = ", ".join(m_i.id + "->" + m_j.id
```

```
                for m_i, m_j in self.dg)
        return "modules: {0}, tests: {1}".format(mods, tests)
```

Při rozboru si pozornost zaslouží tři rysy implementace.

Za prvé je to použití kolekcí pro úschovu podřízených objektů. Moduly jsou překvapivě uloženy ve dvou kolekcích zároveň – seznamu a slovníku. Důvod je jednoduchý: dvojí identifikace modulů. Při použití identifikace podle interního identifikátoru je vhodnější slovník (mapuje interní identifikátory na moduly), při použití indexů je výhodnější seznam (zachovává pořadí vložených modulů). Obě operace tak lze implementovat s časovou složitostí $O(1)$. Alternativou by bylo použití kolekce *collections.OrderedDict* ta však plně podporuje pouze rozhraní slovníku nikoliv seznamu (především nepodporuje indexaci). Navíc i ta je interně implementována jako spojení pole a hashovací tabulky (zvýšení paměťové náročnosti není velké, v obou datových strukturách jsou uloženy jen reference na položky).

Atomické kontroly jsou uloženy v podobě množiny, jejíž položky jsou jsou specializované objekty. Atomické kontroly nemusí být uspořádané, mezi dvěma moduly může být jen jedna atomická kontrola (= v množině stačí jen jedna instance).

Druhým zajímavým rysem je zadávání této množiny tj. zároveň i (orientovaných) hran dignostického grafu. To by se mělo dít rychlým a zároveň přehledným zápisem. Naše implementace podporuje zápis *id1-id2* (kde id1 a id2 je interní identifikátory uzlů). Tento zápis je interpretován jako hrana z modulu s identifikátorem *id1* do modulu *id2*. Zápis lze zkrátit zřetězením typu 1-2-3 nebo rozvětvením 1-2,3 (hrana z modulu 1 do 2 a zároveň z 1 do 3) resp. „světvením“ (1,2-3 = hrana z modulu 1 do 3 a z 2 do 3). Méně přehledné jsou různé kombinace jako je 1,2-3,4 (hrany 1-3; 1-4; 2-3; 2-4) nebo 1-2,3-4 (hrany 1-2; 1-3; 2-4; 3-4). Jednotlivé specifikace se oddělují středníkem. Parsování tohoto vstupního mikrojazyka je relativně snadné (tříúrov-

ňové dělení řetězce pomocí metody *split*), je je však poněkud nerobustní (nepočítá s mezerami, bezprostředně sousedícími oddělovači, apod.)

Posledním pozornostihodným rysem programu je representace dvourozměrného pole, do něhož je ukládána matice sousednosti diagnostického grafu. Python podporuje přímo jen jednorozměrné seznamy. Dvourozměrné pole lze sice representovat pomocí pole polí (jako v Javě, či zubatá pole v C#), není to však příliš efektivní a bezpečné (takovéto pole je ve skutečnosti referencí na pole referencí, odkazující pole referencí na objekty čísel)

Otázka: Velikost libovolného objektu v Pythonu lze zjistit pomocí metody sys.getsizeof. Kolik bytů je potřeba na representaci malého číselného objektu a odkazu na něj? (výsledek závisí na tom zda používáte 32-bitový či 64-bitový systém).

Proto využijeme knihovnu *NumPy*, která do Pythonu (kromě jiného) dodává podporu efektivních vícerozměrných polí ve stylu a efektivitě Matlabu

nebo jazyka R. Tato knihovna se používá v téměř všech vědecko-technicky zaměřených pythonských projektech.

Všimněme si především konstrukce pole:

```
matrix = np.matrix(
          np.zeros((self.size, self.size), dtype=np.int8),
          copy=False)
```

Nejdříve je pomocí tovární metody *numpy.zeros* (*np* je běžný alias zavedený zápisem *import numpy as np*) vytvořeno dvourozměrné pole o velikosti, jež je specifikována dvojicí velikosti v obou dimenzích (v pořadí počet-řádku, počet-sloupců). Jeho jednotlivé položky jsou 8-bitové číselné hodnoty (to pro representaci čísel 0 nebo 1 bohatě stačí) a jsou na začátku vynulovány. Toto pole je sice dvourozměrné, prozatím však nepodporuje některé maticové operace jako je maticové násobení, apod. (matice je jen specializací dvourozměrného pole). Proto je zkonstruován nový objekt matice

(*np.matrix*) inicializovaný dříve vytvořeným polem. Objekt matice nealokuje svůj vlastní paměťový blok a nekopíruje do něj čísla z pole, neboť je vytvořen s parametrem *copy=False*. Místo toho nabízí jen jiný přístup k datům původního pole (tzv. *view*, česky průhled).

I když je vytváření systému z již existujících (pojmenovaných) modulů dostatečně obecné (lze například vytvářet heterogenní systém, v němž je každý modul instancí jiné třídy) je pro běžné použití (homogenní systémy s jednoduchými číselnými identifikátory) poněkud předimenzované. Proto je vhodné vytvořit ještě pomocnou metodu, která vytváření systému zjednodušuje (je implementována jako statická metoda třídy *System*).

```
class System
   ...
   @staticmethod
   def generateModules(size, moduleClass, faultyModules):
      '''
```

```
        vytváří seznam modulů s číselnými identifikátory 1..size.
        Parametrem 'moduleClass' by měla být požadovaná třída modulů
        (nebo tovární metoda). Moduly jsou implicitně bezchybné,
        indexy chybných modulů musí být explictně uvedeny v seznamu
        'faultyModules'.
        :type size: int
        :type mvrací oduleClass: executable
        :type faultyModules: list
    '''
    return (moduleClass(
        str(i),
        MState.FAULTY if i in faultyModules else MState.OK)
      for i in range(1, size+1))
```

Otázka: Proč je metoda *generateModules* označená jako statická?

Metoda nevrací přímo nový objekt representující seznam, ale iterátor přes moduly, který lze využít v rámci konstruktoru systému (vrací se iterátor nikoliv seznam, a proto se vyhodnotí lenivě až v konstruktoru).

```
s = System(System.generateModules(8, # počet modulů
                                  PreparatModule, # třída modulů
                                  [1, 3, 7]), # chybné moduly
           "1-5,6;7-8-2-5;4-5-3-7;7-3;4-1;6-8,2-1-7")
           # atomické kontroly
```

Nyní můžeme přistoupit k definici tříd dílčích komponent.

Třída Syndrome

Začneme definici syndromu. Ten je svou podstatou zobrazením množiny atomických kontrol na množinu $\{0, 1\}$. Přirozenou implementací tak je slovník. Základem jsou metody umožňující přístup k jednotlivným výsledkům

(indexace), ověření zda je atomická kontrola obsažena v syndromu (podpora operátoru „*in*“) a iterátor vracející syndrom jako seznam trojic (testující modul, testovaný modul a výsledek), který je vhodný např. pro zobrazení. Pro přidání metody do syndromu slouží metoda *addResult* (jenž naopak očekává representaci atomické kontroly s výsledkem a vkládá je do slovníku).

```
class Syndrome:
    '''
        reprezentace syndromu
    '''
    def __init__(self, system):
        '''
            vytváří prázdný syndrom
        '''
        self.syndrome = {}
```

```
        self.system = system

    def addResult(self, check, result):
        '''
            přidává do syndromu výsledek atomické kontroly.
            Parametrem je atomická kontrola (dvojice modulů)
            a číselný výsledek (0 nebo 1).
            :type check: AtomicCheck
            :type result: int
        '''
        assert check in self.system.dg
        self.syndrome[check] = result

    def __str__(self):
        return ",".join("{0}->{1} = {2}".format(m_i.id,
                                                m_j.id, result)
```

```
                for ((m_i, m_j), result)
                in self.syndrome.items())

    def __getitem__(self, key):
        '''
            operátor indexace, vrací výsledek pro danou atomickou kontrolu
        '''
        return self.syndrome[key]

    def __contains__(self, key):
        '''
            test zda je atomická kontrola obsažena v syndromu. Metoda
            se použije při volání operátoru 'in'
        '''
        return key in self.syndrome
```

```
    def __iter__(self):
        '''
            iterátor přes syndrom. Vrací jednotlivé prvky syndromu jako
            trojice (testující modul, testovaný modul, výsledek)
        '''
        for m_i, m_j in self.syndrome.keys():
            yield (m_i, m_j, self.syndrome[m_i, m_j])
```

Třída *Syndrom*, stejně jako třídy *System* využívá instancí třídy *AtomicCheck* pro representaci atomických kontrol. Atomické kontroly jsou v diagnostickém grafu representovány hranami. V reálné implementaci systému se samodiagnostikou jsou to hardwarové obvody či specializovaný software, jež jsou v obou případech propojeny pomocí obousměrných komunikačních kanálů.

V našem modelu je lze representovat jako uspořádané dvojice modulů a to buď běžné pomocí n-tice (tuple), nebo pomocí malé třídy — přepravky, která sdružuje dvě reference na moduly. Použití n-tic si vyžádá méně úsilí (nic není potřeba definovat a programovat), je však méně přehledné (prvky jsou identifikovány jen indexy). Vlastní třída vyžaduje vytvoření alespoň dvou metod (konstruktor a operátor rovnosti) je však mnohem explicitnější.

Naštěstí existuje standardní alternativa, která výhody obou řešení spojuje — **pojmenované n-tice** (v modulu *collections*). Definice specializované třídy pojmenovaných n-tic vyžaduje jen jedinou řádku:

```
AtomicCheck = namedtuple("AtomicCheck", ("m_i", "m_j"))
```

Tovární metoda *collections.namedtuple* přejímá jméno nové třídy (to je užíváno jen pro dokumentační účely) a sekvenci řetězců s identifikátory komponent. Vrací třídu (třída je v Pythonu také objektem), která je následně spojena s identifikátorem. Tento identifikátor slouží stejně jako identifi-

kátor jakékoliv jiné třídy. Lze jej tudíž volat a tím vytvářen instance (zde konkrétní atomické kontroly)

```
ac = AtomicCheck(m1, m2)
```

Instance pak podporují rozhraní běžné n-tice (např. iterátor či číselnou indexaci), tak i přímý přístup ke komponentám (tj. např. zápis *ac.m_i* apod.).

Všimněte si také použití příkazu *assert*. Ten definuje tzv. aserci tj. kontrolu, zda program pracuje v souladu s programátorovým předpokladem, tj. např. zda jsou metody volány se očekávanými hodnotami parametrů, resp. zda jsou výsledky v rozumných mezích, apod. Je to jakýsi kontrakt. Pokud je aserce splněna, tj. podmínka uvedená za ní je pravdivá; pak program může pokračovat (a pokusí se fungovat správně), jinak odmítne nést odpovědnost a skončí (pomocí speciální výjimky včetně zobrazení nesplněné aserce).

Aserce mají podobnou funkci jako výjimky, resp. tzv. testování jednotek (unit testing). Výjimky se používají v případě, kdy chybnou funkci programu způsobují situace, které nelze odstranit laděním programu (nedostatek prostředků, chybný vstup), resp. když program není schopen danou situaci vyřešit (i když je evidentně správná). Testování jednotek řeší spíše logické chyby programů, tj. situace, kdy je sice stav programu obecně akceptovatelný, ale v daném okamžiku neodpovídá požadované funkčnosti. Pokud například přidáme do prázdného košíku v e-shopu výrobek a on se v něm neobjeví, pak je to logická chyba (požadavek byl jiný), ale samotný stav prázdného košíku není chybový či výjimečný (např. zobrazení musí fungovat) tj. nelze použít aserci ani výjimku.

Zde je ověřeno, že přidaný výsledek odpovídá existující atomické kontrole daného systému. Pokud tomu tak není, je to buď programátorská chyba při jeho generování (tj. přímo v kódu dané metody), nebo měl být zachycen

dříve (pokud by se výsledky četly z externího vstupního zařízení) a měla být vyvolána výjimka.

Otázka: Aserce se většinou eliminují, pokud program běží v tzv. *release* režimu (= když běží u zákazníka). Podle dokumentace: *The current code generator emits no code for an assert statement when optimization is requested at compile time* (optimalizován pomocí volby -O by měl být jen *release* kód). Jaké to má výhody a nevýhody?

Třída Module

Nyní již zbývá implementace poslední třídy — třídy, jejíž instance representují moduly systému. Protože chování modulů v procesu samokontroly není předem jasně definováno (záleží na typu modelů, použité abstrakci apod.), vytvoříme nejdříve abstraktní třídu.

```
class MState(Enum):
    '''
        výčtový typ podporovaných stavů systémových modulů
    '''
    OK = 0
    FAULTY = 1


class Module:
    '''
        reprezentace modulu v rámci systému (abstraktní třída)
    '''
    def __init__(self, ident, initialState=MState.OK):
        '''
            konstruktor modulu
```

```
        ident: vlastní identifikátor modulu
        :type initialState: MState
    '''
    self.state = initialState
    self.id = ident

@property
def faulty(self):
    return self.state == MState.FAULTY

def atomicControlOn(self, otherModule):
    '''
        atomická kontrola prováděná modulem 'self'
        na modulu 'otherModule'.
        V bázové třídě je tato metoda abstraktní (obecně mohou
        existovat různé modely atomické kontroly).
```

```
            :type otherModule: Module
        '''
        raise NotImplementedError("abstract method")

    def __rshift__(self, otherModule):
        '''
            Namísto zápisu m.atomicControlOn(b) lze použít
            syntaktické zkratky a >> b.
        '''
        return self.atomicControlOn(otherModule)
        # delegováno na původní metodu

    def __str__(self):
        return self.id + "(" + str(self.state) + ")"
```

Abstraktní třída implementuje pouze společnou funkčnost všech modulů, schopnost nést identifikátor a vnitřní stav. Vnitřní stavový prostor je navíc jen dvouprvkový — modul je bezchybný (*ok*) nebo selhává s detekovatelnou chybou (*faulty*). Tato množina je representována výčtovou třídou *MState* (explicitní výčtové třídy jsou v Pythonu novinkou od verze 3.4).

Konstruktor je tak triviální a jednoduchá je i metoda pro zapouzdření aktuálního stavu (je to metoda *faulty* vracející *True*, jen tehdy, když je stav modulu roven *MState.FAULTY)*.

Vlastní funkce modulu v rámci samokontroly je určena metodou *AtomicControlOn*, která vrací výsledek atomické kontroly prováděné daným modulem (*self*) na jiném modulu v systému (paramer *otherModule*). Tato metoda nemůže být v abstraktní třídě definována, v Pythonu však nemůže zůstat prázdná. Protože existuje určitá pravděpodobnost, že ji někdo omylem zavolá, měla by vždy vyhodit specializovanou výjimku *NotImplementedError*

.

Kromě této metody je definován i operátor pravostranného bitového posunu (>>), který funguje zcela stejně (ve skutečnosti volá výše uvedenou metodu). Je zde jen jako syntaktické pozlátko pro zkrácení zápisu:

```
# klasický zápis (použití metody)
m1.atomicControlOn(m2)
# zápis pomocí operátoru (čteme m1 checks m2)
m1 >> m2
```

Operátor „>>“ není optimální (lepší by byl např. „->“), ale v Pythonu lze předefinovat jen již existující operátory (pro každý existuje speciální metoda). A v nabídce pythonských bohužel není žádný lepší.

Posledním krokem naší implementace je definice konkrétního modulu, který poskytuje atomické kontroly podle Preparatovy abstrakce:

```
def hit(p):
    '''
        vrací 'true' s pravděpodobností 'p'
    '''
    assert 0.0 <= p <= 1.0
    if p == 1.0:
        return True
    return uniform(0.0, 1.0) < p

class PreparatModule(Module):
    def __init__(self, ident, initialState=MState.OK,
                 probabilityOfOneResult=0.5):
        super().__init__(ident, initialState)
        self.p = probabilityOfOneResult

    def atomicControlOn(self, otherModule):
```

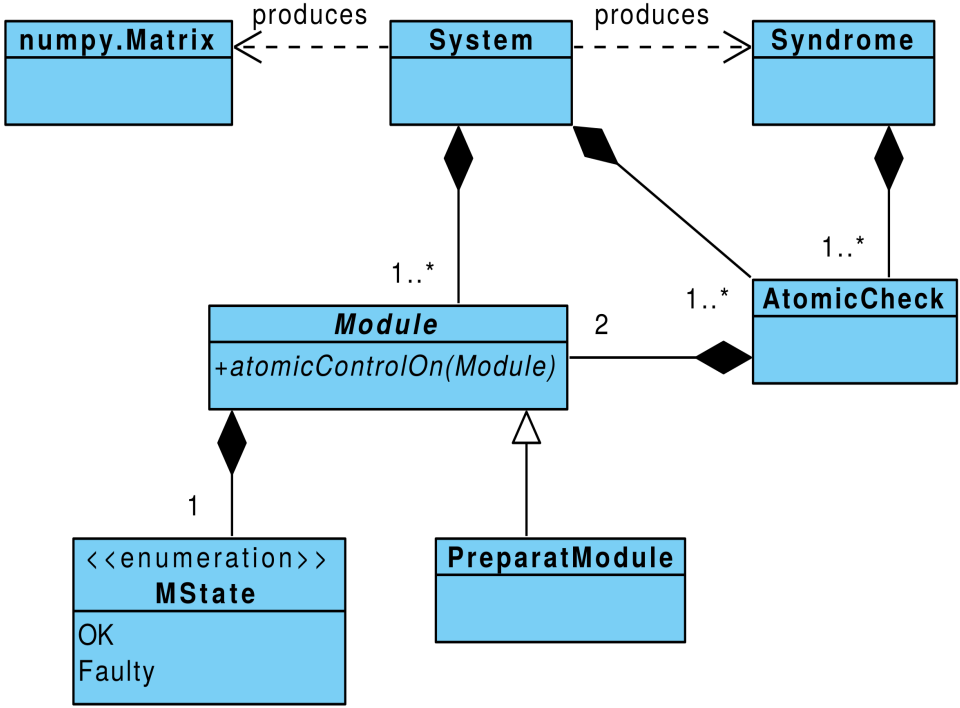
```
        if self.state == MState.OK:
            return 0 if otherModule.state == MState.OK else 1
        else:
            return 1 if hit(self.p) else 0
```

Objekty této třídy, očekávají kromě počátečného stavu i pravděpodobnost výsledku 1 při atomické kontrole prováděné selhávajícím modulem (implicitně je 0.5).

Náhodné nastavení výsledku je prováděno pomocí funkce *hit*, která vrací *true* s danou pravděpodobností. Využívá přitom generátor náhodných čísel s rovnoměrným rozdělením na uzavřeném intervalu [0,1]. Protože pravděpodobnost, že vygenerované číslo je menší než x tj. $P[X < x]$ je v tomto rozdělení rovna x (pro statistiky distribuční funkce rozdělení je $F(x) = x$ pro interval (0,1)), tak stačí vrátit výsledek porovnání náhodně generovaného čísla s požadovanou pravděpodobností vrácení *true*.

Úkol: Proč je speciálně řešen případ s pravděpodobností rovnou 1? (nápověda: podívejte se co přesně vrací funkce *uniform*).

Na závěr implementace se ještě podíváme na třídní UML diagram námi navrženého systému:


numpy.Matrix
produces
System
produces
Syndrome
1..*
1..*
1..*
AtomicCheck
Module
+atomicControlOn(Module)
2
1
<<enumeration>>
MState
OK
Faulty
PreparatModule

3. Grafická presentace diagnostického grafu

- seznámíte se principy vizualizace (oddělení vrstev)
- poznáte základy knihovny *Matplotlib*
- poznáte základní algoritmus pro pozicování orientovaných grafů (*force directed drawing*)

orientovaný graf, grafická primitiva

Presentace diagnostického grafu pomocí výpisu uzlů a hran či pomocí matice sousednosti v zásadě postačuje, avšak není příliš přehledná. Pro rychlou orientaci je vhodnější grafická representace pomocí spojového grafu.

3.1 Návrh

Grafické zobrazení však není tak triviální jako textový výstup, neboť vyžaduje:

1. optimální rozvržení zobrazení uzlů (typicky to jsou kružnice) a hran (šipky tvořené úsečkami nebo složitějšími křivkami). Tuto část lze provést pomocí standardních pythonských knihoven, je to však relativně složitý a matematicky orientovaný problém.
2. použití grafické knihovny pro nakreslení a vizualizaci navrženého grafu

Hlavním cílem při návrhu vykreslovacího kódu je jeho **maximální oddělení** od kódu pro simulaci samodiagnostických systémů, tj. obě části musí být na sobě **maximálně nezávislé**. Tento požadavek je v oblasti programování zcela klíčovým, neboť výrazně usnadňuje udržování a rozšiřování kódu.

V oblasti udržování maximální nezávislost snižuje požadavky na programátory, neboť programátorům stačí znalosti o relativně malé a izolované části kódu (a s ním spojené teorie) spolu se znalostí jednoduchých a stabilních rozhraní. V našem případě tak programátor vytvářející model samodiagnostického systému nemusí znát žádné detaily o kreslení spojových grafů (pouze použije jednoduché rozhraní objektu pro jeho vykreslení). Naopak programátor grafického výstupu nemusí znát model samodiagnostického systému včetně teorie (stavy modulu, atomické kontroly, syndromy apod.).

Výhody se stávají ještě zřetelnějšími s ubíhajícím časem. Každá nezávislá část kódu se sice může dále vyvíjet, avšak pokud obě strany zachovávají rozhraní, pak oba části kódu stále bez problémů spolupracují (bez toho, že by se programátoři domlouvali či spolupracovali). Při revolučnějším vývoji jedné nebo obou částí, se však může původní společné rozhraní stát úzkým hrdlem, které brání propagaci nových schopností (například grafika, již ne-

ní schopna representovat rozšířené stavy systému resp. vice versa model systému nemusí být schopen využít rozšířené možnosti zobrazení).

I zde však existuje relativně bezbolestné řešení (relativně vůči přepsání celého vzájemně provázaného kódu). Rozhraní lze rozšířit (při zachování původné funkčnosti) nebo dokonce zcela změnit (a to i jednostranně, druhá strana však v tomto případě musí vytvořit tzv. adaptér převádějící jedno rozhraní na druhé).

Oddělení však umožňuje i vlastní nezávislý vývoj a použití obou částí, či jejich výměnu. Například model diagnostického systému může snadno začít využívat zcela jiný kód pro kreslení grafů (jen za cenu vytvoření nového adaptéru), a kreslení spojových grafů může být použito i ve zcela jiné aplikaci (spojové grafy se nepoužívají jen pro representaci diagnostických grafů). Záměna nemusí být provedena jen v rámci upgradu kódu (tj. v rámci nového překladu), ale může být realizována i za běhu aplikace (tj. dynamic-

ky).

Tento aspekt oddělení nezávislých částí kódu je hlavním důvodem pro další úroveň oddělení, tentokrát přímo v kódu pro vykreslování spojových grafů. Část zabývající se hledáním optimálního rozvržení silně závisí na zvoleném polohovacím algoritmu, nezávisí však na použité grafické knihovně (neprovádí žádnou kreslící operaci). Naopak kód pro vlastní vykreslování je zcela závislý na použité knihovně (existují desítky těchto knihoven pro různé platformy). Obě části jsou značně nezávislé a při vhodně zvoleném rozhraní lze zajistit snadnou zaměnitelnost na obou stranách. Potenciálně tak lze kombinovat různé algoritmy pro pozicování s různými grafickými výstupy.

Klasický OOP prostředek zajišťující oddělení kódu je *zapouzdření* tj. soustředění souvisejících metod a dat do objektů, přičemž jen některé metody jsou použitelné v objektech jiných tříd (ty tvoří rozhraní). Zapouzdření může být doplněno využitím polymorfismu, který umožní statickou i dynamickou

záměnu bez změny na úrovni kódu (v Pythonu je polymorfismus automatický, u staticky typovaných jazyků je však nutno použít explicitního sdíleného rozhraní nebo dědičnosti).

Existuje však zajímavá alternativa, pokud je rozhraní tvořeno jen jedinou metodou. V tomto případě lze namísto vytváření třídy **zapouzdřit funkcionalitu do běžné funkce** (resp. libovolné entity, která se jako funkce chová). To je vhodné především v případě, kdy funkce transformuje data na vstupu na jiná data na výstupu.

V našem případě funkce pro rozvržení přijímá vhodnou representaci spojového grafu (např. matici sousednosti) a vrací optimální 2D polohy zobrazení uzlů. Pro jednoduchost předpokládáme, že hrany budou přímé spojnice. Navíc se při optimalizaci neberou v potaz popisky uzlů a hran (ty nejsou v této fázi dokonce ani k dispozici).

Zobrazovací funkce přijímá representaci spojového grafu, popisky uzlů resp. hran a samozřejmě i polohy uzlů a vrací nakreslený graf v podobě bitmapy ve formátu PNG. I zde je vidět jisté zjednodušení, neboť je podporován jen jeden formát bitmapy (dodatečný převod mezi formáty je relativně snadný).

Pro optimální spolupráci obou komponent i jejich interakci s okolím je nutné zvolit optimální formát pro přenos dat. Tento formát by měl splňovat tři základní podmínky:

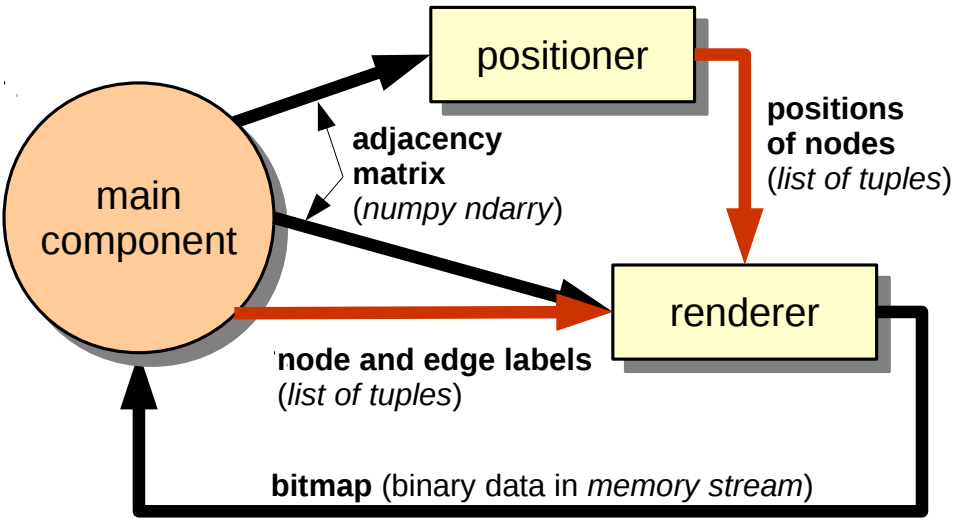
1. **výměnný formát musí být relativně jednoduchý a neměl by využívat interní třídy, které nemusí být k dispozici tvůrcům spolupracujících komponent resp. jsou příliš těžkotonážní (= obsahují zbytečné datové členy a metody). Optimální je využití základních kolekcí. Alternativně lze využít externí textový formát (JSON, Graphviz DOT), což však vyžaduje použití parseru.**

2. **výměnný formát by měl být efektivní tj. nenáročný na paměť a rychle zpracovatelný. Z tohoto hlediska jsou výhodnější vestavěné kolekce než externí formáty. Toto hledisko bylo zohledněno především u hlavního vstupu (matice sousednosti v podobě *numpy dvojrozměrné pole*) a výstupu (*komprimovaná standardní bitmapa*)**

3. **výměnné formáty by měly mít přímou podporu ve standardních pythonských knihovnách (to platí pro standardní kolekce) resp. alespoň v běžně dostupných knihovnách (= publikovaných na PyPI)**

Na základě těchto požadavků byly jako základní interní formát zvoleny pojmenované *n-tice* s jednoduchou strukturou. Výhodou je snadné zpracování a snadná rozšiřitelnost (lze přidávat další položky bez narušení zpětné kompatability a nutnosti používat číselné indexy). Pojmenované n-tice se také snadno definují. Pro přenos bitmapy, která nemá standardní všeobecně preferovaný typ (= třídu podporovanou více než jednou knihovnou) je

jako úložiště binárních dat použit paměťový binární proud (= oblast paměti, do které lze číst a zapisovat jako do souboru), jenž je v Pythonu dostupný pomocí instance třídy ***io.BytesIO***.

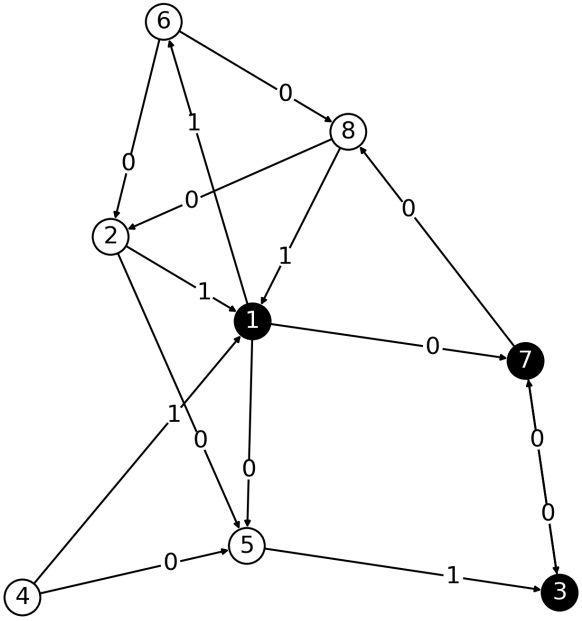


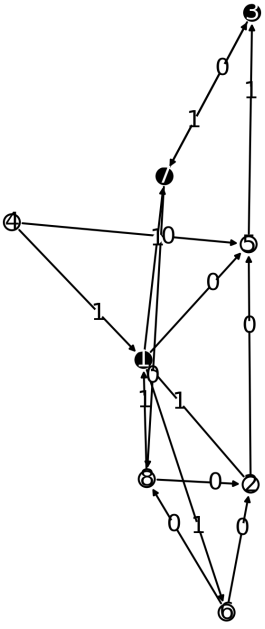
3.2 Implementace

Nyní již k vlastní implementaci. Pro pozicování byl použit algoritmus označovaný *force directed drawing*, který používá pro kreslení grafů relativně jednoduchý fyzikální princip. Uzly jsou nejdříve zcela náhodně rozmístěny. Pak se simuluje jejich pohyb způsobený dvěma silami:

1. všechny body jsou navzájem odpuzovány silou, která klesá postupně s vzdáleností (je běžně označována jako *antigravitace*, i když na rozdíl od gravitace klesá se vzdáleností lineárně)
2. body které jsou spojeny hranou (bez ohledu na její orientaci) jsou k sobě přitahovány silou, která naopak se vzdáleností lineárně roste. Přibližně modeluje situaci, kdy jsou příslušné uzly spojeny pružinou (angl. *spring*). Čím je pružina delší tím jsou uzly k sobě více přitahovány.

Po určité době dojde k rovnováze mezi oběma silami a uzly zaujmou pevnou polohu. Pokud byly počáteční polohy rozumné pak, je výsledek poměrně optimální. Uzly jsou od sebe dostatečně vzdáleny (ty s menší koincidencí jsou o něco vzdálenější), průsečíky mezi hranami jsou relativně málo četné (i když nejsou přímo optimalizované). Výsledkem však může být i mnohem horší (pokud jsou počáteční podmínky horší). Ve skutečnosti je nutné spustit generování vícekrát dokud nezískáte ucházející graf (u testovacího grafu asi 2-5 krát). Ukázky dvou výstupů jsou na následujících dvou obrázcích.


6
8
2
1
7
5
4
3
0
1
0
0
0
1
0
1
0
0
1
0
0
0
1


5
0
1
1
4
1
0
5
1
0
0
1
1
1
2
0
0
1
0
6

V implementaci pozicování začneme definicí pojmenovaných n-tic, které uchovávají informace při přenášení mezi komponentami (včetně hlavního programu). Representace pozice je zřejmá (dvojice *x*,*y*). Atributem uzlu je jeho popisek (*id*) a barva výplně (*bgColor* = background color) a textu (*fgColor* = foreground color). Atributem hrany je index počátečního a koncového uzlu (ty hranu identifikují v matici sousednosti) a popisek (označený jako *text*). V našem případě bude textem výsledek atomické kontroly.

```
Position = namedtuple("Position", ("x", "y"))
NodeAttribute = namedtuple("NodeAttribute",
                    ("id", "bgcolor", "fgcolor"))
EdgeAttribute = namedtuple("EdgeAttribute",
                    ("start", "stop", "text"))
```

Dále následuje definice pomocných interních tříd, které representují základní objekty analytické geometrie na ploše (bod a vektor) včetně základ-

ních operací (jsou pro ně předefinovány operátory, aby se lépe používaly).

```
class Point:
    '''
    Bod ve 2D prostoru.
    '''
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def sqdist(self, other):
        '''
        Druhá mocnina vzdálenosti mezi body self a 'other'.
        '''
        return ((self.x-other.x)*(self.x-other.x)
              + (self.y-other.y)*(self.y-other.y))
```

```
def __sub__(self, startPoint):
    '''
    Operace odečítání bodů. Vektor z bodu startPoint do self.
    '''
    assert isinstance(startPoint, Point)
    return Vector(self.x - startPoint.x,
                  self.y - startPoint.y)

def __add__(self, vector):
    '''
        Operace sečtení bodu a vektoru.
        Bod získaný posunem bodu self o vektor 'vector'.
    '''
    assert isinstance(vector, Vector)
    return Point(self.x + vector.x, self.y + vector.y)
```

```
    def toTuple(self):
        '''
            Převod bodu na n—tici souřadnic (x,y)
        '''
        return (self.x, self.y)


class Vector:
    '''
        Vektor ve 2D prostoru.
    '''
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
def __rmul__(self, scalar):
    '''
    Násobení vektoru skalárem. Operand třídy vektor je druhým
    (pravým) operandem (je použita speciální metoda 'rmul'
    nikoliv 'mul').

    levý operand:
    :type scalar: numbers.Real
    '''
    assert isinstance(scalar, numbers.Real)
    return Vector(scalar*self.x, scalar*self.y)

def __add__(self, vector):
    '''
    Sčítání vektorů.
```

```
        :type vector: Vector
        '''
        assert isinstance(vector, Vector)
        return Vector(self.x + vector.x, self.y + vector.y)
```

Poslední pomocnou třídou je representace uzlu grafu pro účely algoritmu pozicování — *Vertex*. Uzel grafu pro tento účel kromě pozice obsahuje i vektor rychlosti a síly užívané při simulaci jeho pohybu.

```
class Vertex:
    '''
    Vrchol grafu pro algoritmus pozicování řízený simulovanými silami
    (force—directed).
    '''
    def __init__(self):
        # náhodná pozice s rovnoměrným rozdělením
```

```
        self.pos = Point(uniform(0, 300), uniform(0, 300))
        self.velocity = Vector(0.0, 0.0) # rychlost
        self.force = Vector(0.0, 0.0) # síla
```

Funkce pro polohování vektorů je při použití objektů třídy *Point* a *Vector* relativně stručná a přehledná:

```
def simplePositioner(dg):
    '''
        Funkce pro pozicování využívající jednoduchý algoritmus
        řízený fyzikálnámi pseudosilami.

        Jediným parametrem je matice sousednosti.
        Převzato a upraveno z
        http://blog.ivank.net/force-based-graph-drawing-in-as3.html
        (originálně v ActionScript 3)
```

```
    :type dg: numpy.matrix
    '''
    assert dg.shape[0] == dg.shape[1] # jen čtver. matice
    size = dg.shape[0] # počet uzlů
    # generuj 'size' náhodně rozmístěných uzlů/vrcholů grafu
    vertices = [Vertex() for i in range(size)]
    # proveď (size^2)/2 výpočtů poloh
    for i in range(size * size // 2):
        # pro každý vrchol 'v' v řádku matice a jeho index 'i'
        for i, v in enumerate(vertices):
            # nastav počásteční nulovou sílu
            v.force = Vector(0.0, 0.0)
            # a pro každý vrchol 'u' ve sloupci matice a jeho index 'j'
            for j, u in enumerate(vertices):
                # s výjimkou identity vrcholů 'u' a 'v'
                if u is v:
```

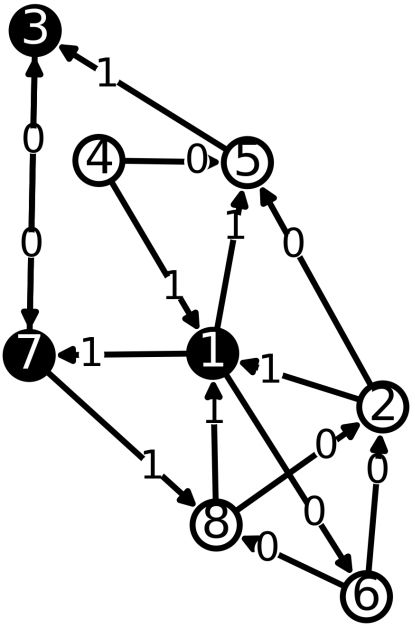
```
                continue
            # vypočítej čtverec vzdálenosti mezi vrcholy 'u' a 'v'
            rsq = u.pos.sqdist(v.pos)
            # vypočti a vektor. přičti odpudivou sílu mezi 'u' a 'v'
            v.force += (200/rsq)*(v.pos - u.pos) #antigravitace
            if dg[i, j] or dg[j, i]: # jsou—li uzly spojeny hranou
                # vektorově přičti přitažlivou sílu mezi vrcholy
                v.force += 0.06*(u.pos-v.pos) # obdoba pružin
        # poté změn vektor rychlosti přičtením síly (integrace
        # zrychlení získaného podle vzorce a = F/m, kde m = 1)
        # se zohledněním útlumu
        v.velocity = 0.85*(v.velocity + v.force)
    # nakonec u každého vrcholu
    for v in vertices:
        # vypočti novou pozici přičtením vektoru rychlosti
        v.pos += v.velocity
```

```
    # vrať konečné pozice vektorů jako seznam pojmenovaných n—tic
    return [Position(vertex.pos.x, vertex.pos.y)
            for vertex in vertices]
```

Otázka: Co nastane v případě grafu s izolovaným uzlem (= uzlem z něhož resp. do něhož nevede hrana?

Pro druhou část zobrazovací kompenenty (renderer) byla zvolena knihovna *matplotlib*. Ta nabízí vyšší úroveň abstrakce než běžné kreslící knihovny (umí například přímo kreslit šipky)

Renderovací komponenta má i speciální podporu pro slabozraké, Při běžném nastavení (viz předchozí obrázky) může být graf obtížněji čitelný. Proto je v renderovací části implementována konstanta, která umožní zvětšit uzly, všechny linie a také písmo. Výsledek při dvojnásobném zvýraznění je vidět na následujícím obrázku (je bohužel mnohem těžší vygenerovat čitelný graf při polohování, neboť dílčí prvky se mnohem více ovlivňují).


3
1
0
4
0
5
0
1
0
1
7
1
1
1
2
1
0
0
8
0
0
6

Rozsáhlost renderovací části (téměř 100 řádků) činí použití jediné funkce dost nepraktickým. Funkci lze samozřejmě rozdělit do více podfunkcí to si však vyžádá předávání relativně velkého množství pomocných parametrů (ty se týkají především rozsahu zobrazovací plochy).

Naštěstí i zde nabízí Python alternativu. Namísto funkce lze využít tzv. volatelný objekt. Volatelný objekt:

1. **lze volat stejně jako funkci a včetně předání parametrů. Navíc se používá stejná syntaxe tj. zápis *object(parametry)*.**
2. **volatelný objekt je však na rozdíl od funkce explicitně instanciován a inicializován konstruktorem své třídy. To umožňuje předat počáteční parametry, které se nemění při každém volání volatelného objektu (konstruktor tak slouží jako parametrizovatelná továrna na funkce). My to využijeme pro předání konstanty pro zvětšení tloušťky čar a velikost písma (ta tak nemusí být předávána spolu s vlastnostmi grafu**

při každém volání).

3. třída volatelného objektu může kromě vlastní přímo volatelné metody/funkce definovat i další veřejné či soukromé (pomocné) metody. Ty mohou sdílet společný stav pomocí atributů objektu.

Volatelný objekt se od běžného objektu liší jen v tom, že definuje speciální metodu ***__call__***. Tato metoda je volána, je-li objekt použit jako funkce. V definici naší renderovací třídy je tato metoda definována hned za konstruktorem a interně volá jednotlivé dílčí kreslící (neveřejné) metody.

```
from matplotlib.backends.backend_agg import FigureCanvasAgg
from matplotlib.figure import Figure
from matplotlib.patches import (Ellipse, FancyArrowPatch,
                        Circle)


class MatplotRenderer:
    '''
```

```
    Třída poskytující volatelný (callable) objekt použitelný
    jako renderovací (zobrazovací funkce).
'''
def __init__(self, blackCoefficient=1.0):
    '''
        blackCoeficient =
        koeficient ovlivňující velikost uzlů, textu a šířku hran.
        Čím je větší, tím jsou tyto prvky výraznější (a lépe čitelné).
        Výsledek však může být méně čitelný.
    '''
    self.bc = blackCoefficient

def __call__(self, positions, dg, nodeAttrs, edgeAttrs):
    '''
        Volání volatelného objektu
        (navenek se objekt jeví jako funkce).
```

```
        Přejímá polohy uzlů (positions), matici sousednosti (dg),
        a seznamy atributů uzlů a hran.

        :type positions: list
        :type dg: numpy.matrix
        :type nodeAttrs: list
        :type edgeAttrs: list
    '''
    self.positions = positions
    self.dg = dg
    self.size = len(positions)
    assert self.size == self.dg.shape[0]
    assert self.size == self.dg.shape[1]
    assert self.size == len(nodeAttrs)

    self.nodeAttrs = nodeAttrs
```

```
        self.edgeAttrs = edgeAttrs

        self._init()
        self._drawEdges()
        self._drawNodes()
        return self._finish()

    def _init(self):
        # vytvoření prázdného obrázku v matplotlib (rozměr 5x5 palců
        # při rozlišení 300 dpi tj. minimálním rozlišení tiskáren)
        self.fig = Figure(figsize=(5, 5), dpi=300,
                          tight_layout=True, facecolor="white")
        # do něhož je přidána jedna soustava souřadnic
        # soustavy jsou umístěny v mřízce (zde je to mřížka 1x1)
        self.ax = self.fig.add_subplot(1, 1, 1)
```

```
        # výpočet krajních souřadnic
        self.xmin = min(node.x for node in self.positions)
        self.ymin = min(node.y for node in self.positions)
        self.xmax = max(node.x for node in self.positions)
        self.ymax = max(node.y for node in self.positions)

        # výpočet průměru kružnic pro uzly
        self.d = self.bc * max(self.xmax-self.xmin,
                               self.ymin-self.ymax) / 15.0

    def _drawEdges(self):
        # pro každý i—tý uzel
        for i in range(self.size):
            # kombinovaný s j—tým uzlem
            for j in range(self.size):
                # je—li hrana z i—tého do j—tého
```

```
            if self.dg[i, j]:
                # přidej šipku z pozice posA do posB
                # se stylem 'arrowstyle' (vyplněný trojúhelník)
                # špička zvětšená 25x oproti interním souřadnicím
                # šipka je na obou koncích zkrácená o 0.75*d
                # (končí na na vnějším obvodu uzlu)
                # má danou barvu (černou)
                # a šířku 1.5 * koeficient tučnosti (lw = line width)
                self.ax.add_patch(FancyArrowPatch(
                            posA=self.positions[i],
                            posB=self.positions[j],
                            arrowstyle="-|>",
                            mutation_scale=32*self.bc,
                            shrinkA=0.75*self.d,
                            shrinkB=0.75*self.d,
                            color="black",
```

```
                        lw=1.5*self.bc))

    # přidej krátký popisek (jeden znak)
    for start, stop, text in self.edgeAttrs:
        # pozice počátečního uzlu
        startPoint = Point(self.positions[start].x,
                           self.positions[start].y)
        # pozice koncového uzlu
        endPoint = Point(self.positions[stop].x,
                         self.positions[stop].y)
        v = endPoint - startPoint
        # pozice popisku (ve 2/3 vzdálenosti ke koncovému uzlu)
        # je tak zřejmé k jakému směru patří překrývají se šipky
        labelPoint = startPoint + 0.66*v
        # podklad (bílá kružnice), zakrývá část šipky
        self.ax.add_patch(Circle(
```

```
                        (labelPoint.x, labelPoint.y),
                        self.d/4, color="white"))
        # text popisku vycentrovaný na pozici 'labelPoint'
        item = self.ax.text(labelPoint.x, labelPoint.y,
                            text, ha="center", va="center")
        # nastavíme velikost písma (implicitně je 10 bodů)
        item.set_fontsize(10*self.bc)

    def _drawNodes(self):
        # prochází zároveň seznam pozic uzlů a odpovídající atributy
        # 'zip' vrací seznam dvojic (pozice, atribut) pro odpovídající
        # pozice seznamů
        for node, attr in zip(self.positions, self.nodeAttrs):
            # přidá kružnici s průměrem d
            # (= elipsa se stejnou výškou a šířkou)
            # a požadavanou barvou výplně 'facecolor'
```

```
            self.ax.add_patch(Ellipse(node,
                                      width=self.d, height=self.d,
                                      facecolor=attr.bgcolor,
                                      lw=1.5*self.bc))
            # dovnitř kružnice nakreslí centrovaný text —— popisek uzlu
            # s barvou 'color'
            item = self.ax.text(node.x, node.y, attr.id,
                                ha="center", va="center",
                                color=attr.fgcolor)
            # nastavíme velikost písma (implicitně je 12 bodů)
            item.set_fontsize(12*self.bc)

    def _finish(self):
        # nastaví limit v ose x
        # (je započítán i poloměr kružnice krajních uzlů + šířka obvodu)
        self.ax.set_xlim(self.xmin - self.d/2 - 3,
```

```
                self.xmax + self.d/2 + 3)
        # totéž pro osu y
        self.ax.set_ylim(self.ymin - self.d/2 - 3,
                self.ymax + self.d/2 + 3)
        # je nastaven poměr jednotky na ose x a y (aspect ratio) na 1
        # = uzly budou skutečně kružnice
        self.ax.set_aspect(1)
        # je vypnuto zobrazení os (souřadnice jsou u spojového grafu
        # bezvýznamné)
        self.ax.axis("off")
        # je vytvořen paměťový proud (= oblast paměti do níž lze
        # zapisovat jsko do souboru)
        memoryStream = io.BytesIO()
        # je vytvořena kreslící plátno do něhož budou zakresleny
        # všechny dříve vytvořené grafické objekty
        # Agg canvas je optimalizován pro výstup v PNG bitmapě
```

```
        # viz http://en.wikipedia.org/wiki/Anti-Grain_Geometry
        canvas = FigureCanvasAgg(self.fig)
        # canvas je renderován uložen do otevřeného proudu
        canvas.print_png(memoryStream,
                         facecolor=self.fig.get_facecolor())
        # ukazatel aktuální pozice v proudu je nastaven na začátek
        # proud je tak připraven na následné čtení
        memoryStream.seek(0)
        return memoryStream
```

Otázka: Jaké jsou výhody volatelného objektu oproti běžné funkci?

Renderovací a polohovací lze volat přímo z nové metody třídy *System* (mohla by mít například jméno *toBitmap*). I to však vnáší zbytečnou závislost kódu simulačního (= třídy *Syste*m, apod.) na zobrazovacím kódu a jeho rozhraní. Proto je vhodné vytvořit další třídu, která bude oba světy spojovat.

Tato třída bude realizovat dva vzájemně provázané návrhové vzory:

1. **most — neboť bude oddělovat protokol zobrazovacích funkcí od simulační komponenty (s možností snadného přechodu na jiné rozhraní pro zobrazování)**
2. **fasáda — zakrývá detaily zobrazovacího protokolu (dvě spolupracující funkce) a provádí převod interní representace modulů na datový protokol očekávaný zobrazovačem (matice sousednosti a jednoduché seznamy n-tic). Konstruktor přejímá jen objekt zobrazovaného systému a jeden další (nepovinný) parametr — koeficient šířky čar a velikosti textu (*blackCoefficient*). Pro vytvoření bitmapy stačí zavolat jedinou bezparametrickou metodu nad vytvořeným zobrazovačem (*process*).**

```
class DGViewer:
    '''
```

```
    Zobrazovač diagnostických grafů v podobě orientovaného
    spojového grafu.

    Uzly rozmisťuje funkce 'positioner', kreslení do PNG
    bitmapy provádí funkce 'renderer'.
    :type system: selfdiagnosis.System
    :type syndrome: selfdiagnosis.Syndrome
'''
def __init__(self, system, syndrome=None,
             blackCoefficient=1.0):
    self.system = system
    self.syndrome = syndrome
  # skutečná funkce
    self.positioner = simplePositioner
  # volatelný objekt
    self.renderer = MatplotRenderer(blackCoefficient)
```

```
        assert (not self.syndrome or self.system
                  == self.syndrome.system)

    def _positioning(self):
        # převod diagnostického grafu do vstupního formátu
        # a volání pozicovací funkce (bridge)
        self.nodePoints = self.positioner(
                      self.system.toAdjacencyMatrix())

    def _toBitmap(self, okColor=(1.0, 1.0, 1.0),
                faultyColor=(0.0, 0.0, 0.0)):
        # převod atributů modulu na popisky a vizuální vzhled uzlů grafu
        # stav je převeden na barvu uzlu (OK—bílý, chybný—černý)
        # popisky umístěné v uzlu mají opačné barvy
        nodeAttrs = [
            NodeAttribute(module.id,
```

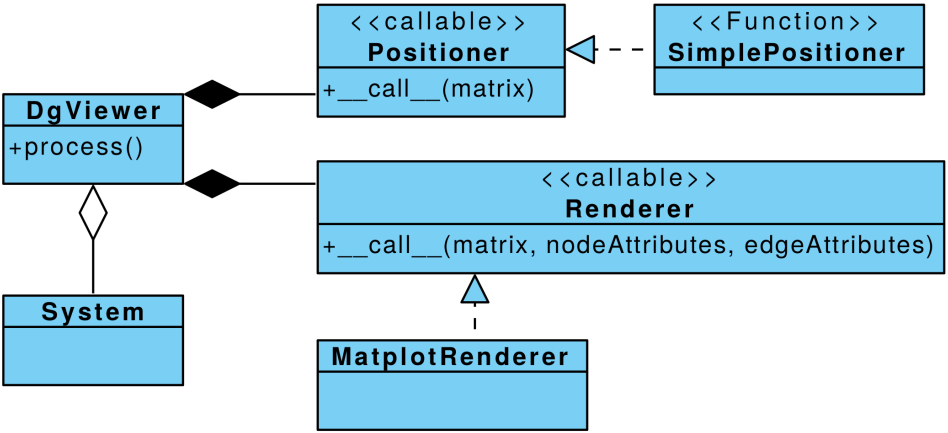
```
            bgcolor=faultyColor if module.faulty else okColor,
            fgcolor=okColor if module.faulty else faultyColor)
        for module in self.system.moduleList]
    # převod dílčích výsledků syndromu na popisky hran
    # popisek je řetezec "0" resp. "1"
    if self.syndrome:  # je—li předán syndrom
        edgeAttrs = [
            EdgeAttribute(self.system.mpos(m_i),
                          self.system.mpos(m_j),
                          str(result))
            for (m_i, m_j, result) in iter(self.syndrome)
        ]
    else:
        edgeAttrs = None
    # volání renderovací (vykreslovací) funkce
    self.bitmap = self.renderer(self.nodePoints,
```

```
                                self.system.toAdjacencyMatrix(),
                                nodeAttrs, edgeAttrs)

    def process(self):
        '''
            převod diagnostického grafu předaného v konstruktoru
            na vizuální representaci v bitmapě. Metoda je fasádou
            zakrývající detaily procesu.

            :return: io.BytesIO
        '''
        self._positioning()
        self._toBitmap()
        return self.bitmap
```

Všimněte si rozdílného zápisu při získání obou zobrazovacích funkcí. Zatímco u polohovací stačí uvést jen jméno funkce, musí být renderovací funkce získána voláním konstruktoru volatelného objektu. Při použití se však již obě entity neliší (chovají se oba jako běžná funkce):

```
self.positioner( ... )
self.renderer( ... )
```

Následující třídní diagram v UML ukazuje výsledný systém tříd. Volatelné objekty jsou definovány pomocí ad hoc vytvořeného stereotypu *callable*, který je obdobou stereotypu *interface*, splní ho však funkce či metody s podporou požadovaných parametrů, návratové hodnoty a kontraktu.



3.3 Použití v GUI

Posledním krokem naší aplikace je zobrazení vytvořených spojových grafů na displeji počítače. Alternativně lze obrázek uložit do grafického souboru, což není příliš složité. Stačí jen na místě paměťového proudu použít bytově

orientovaný proud souborový (otvírá se pomocí vestavěné funkce *open*) nebo lze dokonce jen předat jméno souboru jako řetězec.

```
canvas.print_png("/tmp/graph.png",
                 facecolor=self.fig.get_facecolor())
```

Python nemá vlastní GUI knihovnu, tj. musí používat GUI knihovny napsané v jiných jazycích a frameworcích. Nejčastěji se používá knihovna *Tkinter*, což je pythonská nadstavba knihovny Tcl/Tk. I když byla jednu dobu velmi široce používána (a to nikoliv jen v Pythonu, ale i dalších jazycích), dnes je již trochu zastaralá. Zůstává však jedinou knihovnou zahrnutou ve standardních knihovnách Pythonu tj. měly byste ji mít k dispozici ihned po nainstalování Pythonu (i když ve skutečnosti tomu tak být nemusí).

Modernější GUI knihovny jsou *PyGtk* a *PyQt*, což je jsou pythonské obálky nad klasickými GUI knihovnami. Jsou mnohem modernější, avšak také rozsáhlejší. Komplikovanější je i jejich instalace do Pythonu (i když linuxovské

distribuce ji nabízejí již ve formě balíčků pro systémovou verzi Pythonu).

Zajímavá je také knihovna *Kivy*, která je je moderní a velmi dobře přenositelná (včetně například Androidu).

My pro jednoduchost použijeme knihovnu *Tkinter*.

```
from graphTools import DGViewer
from PIL import Image, ImageTk
from tkinter import Tk, Label
from tkinter.ttk import Notebook

s = System(System.generateModules(3, PreparatModule, [1]),
        "1-2,3;3-2,1")

viewer = DGViewer(s, s.getSyndrome())
image = Image.open(viewer.process())
```

```
root = Tk()
notebook = Notebook(root)
tkImage = ImageTk.PhotoImage(image)
figure = Label(notebook, image=tkImage)
figure.pack()
notebook.add(figure, text="Figure")

notebook.pack()
root.mainloop()
```

Nejdříve je nutno naimportovat několik tříd. Kromě naší třídy *DGViewer* to jsou dva widgety (= vizuální prvky) z knihovny *Tkinter* (kořenový widget Tk a textově-grafický popisek *Label*) a jeden složitější vizuální prvek z rozšiřující knihovny *ttk* (v Pythonu je součástí Tkinteru).

Knihovna *Tkinter* přímo podporuje jen formát GIF (i zde je vidět její stáří). Proto je nutné využít pomocné třídy z knihovny *pillow*, která je nástupcem relativně známé knihovny *PIL*.

Po vytvoření instance třídy DGViewer nad ukázkovým systémem se třemi moduly, vygenerujeme bitmapu v podobě paměťového proudu (naše metoda *DGViewer.process*). Tento proud předáme konstruktoru třídy *PIL.Image*, která vytvoří rozparsovanou representaci bitmapy v paměti.

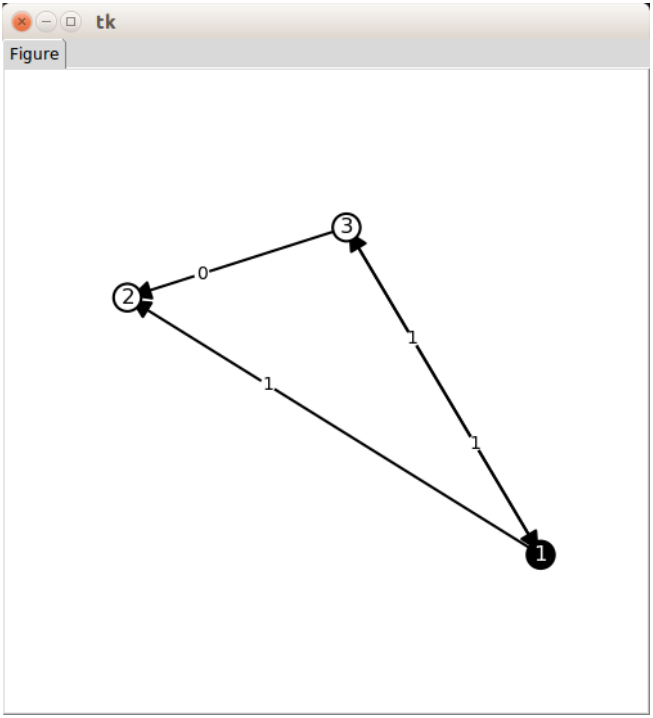
Potom již vytvoříme kořenové GUI okno (= instance třídy *Tkinter.Tk*) a instanci widgetu *Notebook* (prázdný widget, do něhož lze vkládat záložky s ouškem nahoře). Všimněte si, že jeho rodičem (nadřazeným widgetem) je kořenový widget *root*. Pak vytvoříme popisku *label*, do níž vložíme rozparsovanou bitmapu.To však nejde přímo, musíme využít metodu, která transformuje obrázek do formátu podporovaného knihovnou *Tkinter* (je to metoda *ImageTk.PhotoImage*, která vrací objekt třídy *tkinter.PhotoImage*).

Tato metoda je tedy jakýmsi adaptérem.

Aby se v Tkinteru libovolný widget zobrazil, musí být vložen do manažeru rozvržení pomocí metody své metody *pack* (je-li použita bez parametrů, tak se použije standardní rozvržení, které je zde dostatečné).

„Zapakovaný“ widget ještě vložíme jako záložku do notebooku, který také „zapakujeme“.

Pak již stačí spustit na kořenovém widgetu hlavní událostní smyčku (jsme ve světe událostmi řízeného programování) a měli bychom vidět následující okno.


tk
Figure
3
2
1
0
1
1
1

4. Hodnocení možností diagnostiky podle diagnostického grafu

- seznámíte se základní charakteristikami diagnostického grafu
- naučíte se vypočíst pravděpodobnost, že při daném syndromu zjistíte stavy všech modulů
- implementujeme výpočet této pravděpodobnosti (včetně výpočtu diagnostického grafu)

optimální graf, charakteristické číslo grafu

4.1 Teorie

Úspěšnost diagnostiky závisí na počtu selhávajících (chybných) modulů. Tento počet budeme označovat jako t. Pro některé hodnoty t lze vždy vy-

tvořit diagnostický graf , které budou s úplnou jistotou zaručovat správnou diagnostiku (bez ohledu na získaný syndrom).

Pro grafy, u nichž je zaručeno diagnostikování t chybných modulů, tzv. **t-diagnostikovatelnost**, platí následující ***nutná*** podmínka:

V t-diagnostikovatelném grafu musí být modul kontrolován nejméně ***t*** dalšími moduly, přičemž v grafu nejsou vícenásobné atomické kontroly (= hrany).

Pokud však počet chybných modulů ***t*** překročí jisté t_{max}, pak již není možné pro dané t takový diagnostický graf zkonstruovat. Pro toto t_{max} platí: (n je počet uzlů tj. modulů)

$$t_{max} = \left\lfloor \frac{n-1}{2} \right\rfloor \qquad (2)$$

Diagnostický graf **t-optimální**, pokud obsahuje minimální počet hran, které

stačí pro zajištění určité t-diagnostikovatelnosti. Pro hodnotu $t = t_{max}$ je možno graf stručně označit jako optimální.

Počet hran *t-optimálního grafu* lze vypočítat podle vztahu:

$$l = t \cdot n \tag{3}$$

jenž lze v případě *optimálního diagnostického grafu* přechází na:

$$l = t_{max} \cdot n = \left\lfloor \frac{n-1}{2} \right\rfloor \cdot n \tag{4}$$

kde

n **= počet modulů**

t_{max} **= maximální hodnota parametru t**

l **= počet hran optimálního DG**

Otázka: Jaký je nejmenší optimální graf?

Kromě parametru t_{max} a optimálního počtu hran však existují také další *kritéria* pro hodnocení diagnostických vlastností grafu. Například ke každému DG lze přiřadit pravděpodobnost, která bude odrážet diagnostické schopnosti grafu. Konkrétně je to **pravděpodobnost, že syndrom odpovídající určitému diagnostickému grafu umožní správně diagnostikovat stav všech modulů v systému** (dále budeme tuto funkci označovat jako p_{SD} resp. $p_{SD}(DG, p_m)$).

Tato pravděpodobnost kromě diagnostického grafu (= množiny prováděných atomických kontrol) závisí i na pravděpodobnosti, že určitý modul systému selhal tj. je v chybovém stavu. Předpokládáme, že tato pravděpodobnost je pro všechny moduly v systému stejná.

Předpokládejme, že v systému je *k* bezchybných modulů. Pak je pravděpo-

dobnost, že syndrom odhalí stav ostatních $n - k$ modulů rovna:

$$P(A_K) = (1 - p_M)^k \cdot p_m^{n-k} \cdot C_k \qquad \textbf{(5)}$$

p_m = pravděpodobnost, že modul je v chybném stavu (selže). Pravděpodobnost je u všech modulů stejná.

k = počet bezchybných modulů

C_K = počet možností výběru podgrafu tvořeného k uzly, ze kterého jsou všechny ostatní uzly přímo dosažitelné

Výsledek diagnostiky systému je správný, pokud nastane *alespoň jedna* ze situací $A_K,\ k = 1, \ldots, n$. Z toho vyplývá, že pravděpodobnost získání správného výsledku diagnostiky systému na základě syndromu je rovna součtu pravděpodobností $P(A_k)$ pro $k = 1, \ldots, n$:

$$P_{SD} = \sum_{K=1}^{N} P(A_K) = \sum_{K=1}^{N} (1 - P_M)^K \cdot P_M^{N-K} \cdot C_K \qquad (6)$$

Při výpočtu této pravděpodobnosti, je třeba znát hodnoty čísel C_k tzv. charakteristická čísla grafu. Pro relativně malé systémy (řádově méně než deset) lze využít vyčerpávající (úplné) prohledání všech možností pomocí matice sousednosti diagnostického grafu. Tento algoritmus má bohužel exponenciální časovou složitost (je prováděn nad všemi variacemi bez opakování ze všech modulů systému). Těchto variací je $\frac{n!}{(n-k)!}$. Pro n=15 a k=14 je toto číslo 1 307 674 368 000.

Východiskem algoritmus je tzv. rozšířená matice sousednosti diagnostického grafu. Tato matice se liší od běžné matice sousednosti jen tím, že má na diagonále hodnoty 1 (neboť každý uzel grafu je dostupný sám ze sebe).

Následně se charakteristické číslo spočítá podle vzorce (upraven pro py-

thonskou indexaci od nuly):

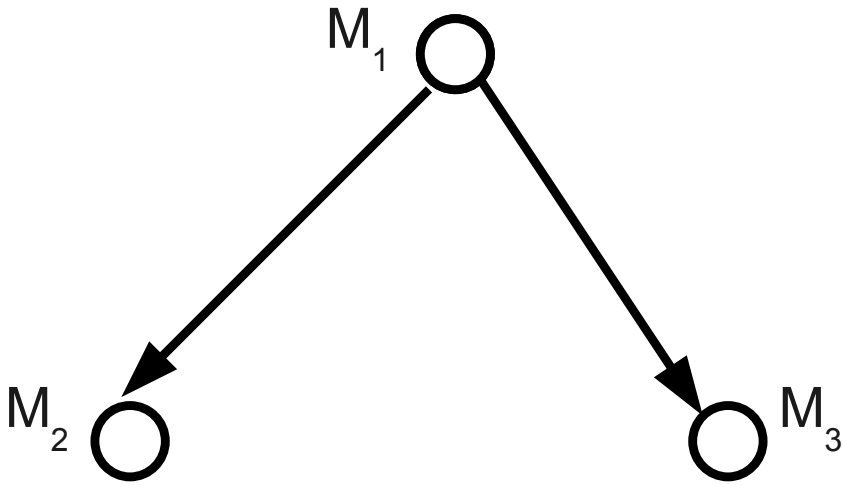
$$C_K = \frac{1}{k!} \sum_{(i_0, i_2, \ldots, i_{k-1}) \in V(k)} \Big[\prod_{j=0}^{n-1} (a_{i_0 j} \oplus a_{i_2 j} \oplus \ldots \oplus a_{i_{k-1} j}) \Big] \qquad (7)$$

kde se suma děje přes všechny k-členné variace indexů $(i_0, i_2, \ldots, i_{k-1})$ bez opakování nad posloupností $0, 1, \ldots, n-1$ (označena symbolem $V(k)$. Produkt se děje přes všechny sloupce matice. Pro každý sloupec se spočítá logický součet všech těch prvků, které leží v daném sloupci a jejichž indexy řádků určuje konkrétní variace (s k prvky). Logický součet odpovídá logické spojce OR, pracuje však s čísly 0 a 1.

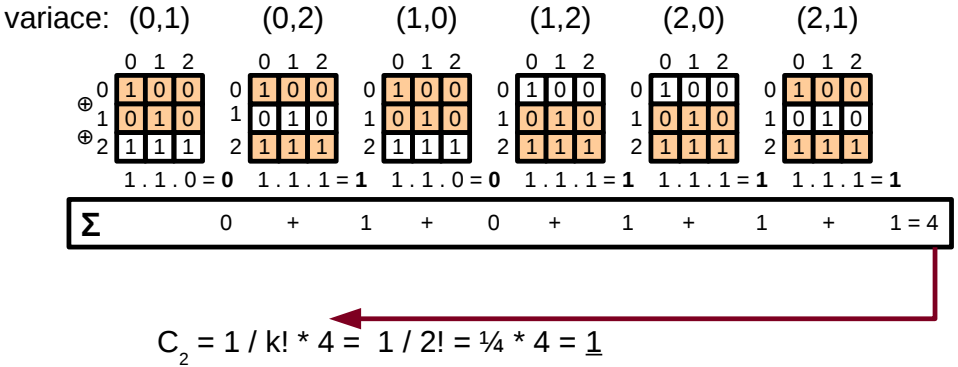
log. součet	0	1
0	0	1
1	1	1

Pro tříprvkové systémy existuje šest dvouprvkových variací indexů: (0,1), (0.2), (1, 0), (1,2), (2,0) a (2,1), tj. sumovat se bude šest hodnot. Každá z těchto hodnot se získá součinem tří čísel (pro každý sloupec), z nichž každé je logickým součtem dvou hodnot, na průsečíku daného sloupce s dvěma řádky, jejichž index je dán danou variací.

Předpokládejme následující diagnostický graf:



Výpočet charakteristického čísla C_2 pro tak malý graf ještě znázornit obrázkem (sčítá se přes šest variací):



4.2 Implementace

Implementace funkcí vracející základní diagnostické charakteristiky diagnostického grafu (t, t_{max} a optimální počty hran) jsou relativně jednoduché. Je zřejmé, že to budou metody třídy *System* (charakterizují celý systém nikoliv modul či syndrom).

```
class System:
  ...
  def ATCount(self):
      '''
          vrací počet všech atomických kontrol v systému
      '''
      return sum(np.nditer(self.toAdjacencyMatrix()))

   def T(self):
      '''
          vrací počet chybných modulů v systému
      '''
      return sum(1 if module.faulty else 0
                 for module in self.moduleList)
```

```
    def Tmax(self):
        '''
            vrací maximální počet modulu, které mohou selhat a graf
            zůstáva (samo)diagnostikovatelný
        '''
        return (self.size - 1) // 2

    def ToptimalATCount(self):
        '''
            vrací minimální počet atomických kontrol, který ještě
            zajistí diagnostiku 't' chybných modulů (kde 't' je aktuální
            počet chybných modulů)
        '''
        return self.T() * self.size

    def optimalATCount(self):
```

```
        '''
            vrací minimální počet atomických kontrol, který ještě
            zajistí diagnostiku 't_max' chybných modulů (kde 't_max'
            je aktuální maximální počet chybných modulů)
        '''
        return self.Tmax() * self.size
```

Počet atomických kontrol v systému (= počet hran v diagnostickém grafu) je zjištěn z matice sousednosti. Pomocí metody *nditer*, která vrací iterátor přes všechny prvky matice (v předem neznámém pořadí), je provedena sumace všech těchto prvků, jenž je pro jednoduchý orientovaný graf rovna počtu hran.

Úkol: Existuje ještě jedna možnost implementace metody *ATCount* (a výrazně jednodušší). Zkuste ji najít a použít.

Při výpočtu t_{max} se využívá celočíselného dělení, které s v Pythonu zapisuje pomocí operátoru „//“. Normální dělení (operátor „/“) může vrátit necelé číslo z množiny $\mathbb{Q} \setminus \mathbb{Z}$ (i pro celočíselné operandy!).

O něco složitější je výpočet pravděpodobnosti p_{SD} (pravděpodobnost správné diagnostiky pro daný diagnostický graf).

Nejdříve si připravíme několik pomocných funkcí (nejsou umístěny v žádné třídě, ale přímo na nejvyšší úrovni programu):

```
import operator
...
def product(iterator):
    '''
        produkt iterátoru (součin všech jeho prvků)
    '''
    return reduce(operator.mul, iterator)
```

```
def fact(n):
    '''
        faktoriál čísla n (n!)
    '''
    return product(range(1, n + 1))


def logadd(a, b):
    '''
        logický součet dvou čísel z množiny {0,1}
    '''
    return 1 if a == 1 or b == 1 else 0


def mathrange(mx):
```

```
    '''
        vrací iterátor od 1 do 'mx' včetně. Vhodnější pro matematickou
        notaci.
    '''
    return range(1, mx+1)
```

Produkt využívá operaci *reduce*, která redukuje iterátor na jedinou skalární hodnotu tím, že mezi jednotlivými prvky aplikuje binární operaci. V případě funkce *product* je touto operací násobení. V Pythonu nelze pro redukci použít přímo operátor, naštěstí modul *operator* poskytuje binární operátory v podobě běžných funkcí — zde je to *operator.mul*.

S pomocí těchto funkcí již můžeme definovat metodu pro výpočet charakteristického čísla C_k diagnostického grafu a metody vracející pravděpodobnost p_{SD}.

```
import itertools

class System:
  ...
  def C(self, k):
      '''
          výpočet charakteristického čísla pro 'k' uzlů
          diagnostického grafu
      '''
      if k == self.size: # pro k == N je charakteristické číslo vždy 1
          return 1
      edg = (self.toAdjacencyMatrix() # rozšířená mat. sousedn.
           + np.identity(self.size, dtype=np.int8))
      suma = 0
      # přes všechny k—prvkové variace s opakování
```

```
        for v in itertools.permutations(range(self.size), k):
            p = product(reduce(logadd,
                               (edg[i, j] for i in v))
                        for j in range(self.size))
            suma += p
        return suma / fact(k)

    def p_sd(self, p_m):
        return sum((1.0-p_m)**k * p_m**(self.size-k)*self.C(k)
                   for k in mathrange(self.size))
```

Implementace výpočtu charakteristického čísla je výrazně zjednodušena podporu různých typů permutací, variací a kombinací v modulu *itertools*. Metoda *itertools.permutation(iterator, k)* vrací nový iterátor, který poskytuje všechny variace bez opakování s velikostí k (v podobě k-tic).

Jméno funkce může být trochu matoucí, neboť v angličtině se nerozlišují variace a permutace, ale že se jedná o požadované variace s opakováním potvrzuje standardní dokumentace:

výsledek: permutations(p, r): r-length tuples, all possible orderings, no repeated elements
příklad: permutations('ABCD', 2) = AB AC AD BA BC BD CA CB CD DA DB DC

Otázka: Zkuste vytvořit funkci, který bude iterátor nad variacemi vracet. Nejjednodušším řešením je generátor iterátoru (= korutina s příkazem *yield*) obsahující vnořené cykly.

S pomocí této funkce je implementace snadná. Nejdříve se vytvoří rozšířená matice sousednosti přičtením jednotkové matice k původní matici sousednosti (obě jsou čtvercové a mají stejný rozměr). V *NumPy* se dává přednost operacím s celými n-rozměrnými poli, před cykly.

Poté následuje cyklus přes všechny k-prvkové variace s opakováním (variace je odkazována proměnou *v*). Poté je vypočítán číslo *p* za pomoci dvou vnořených redukcí. Vnitřní redukce redukuje ty hodnoty v sloupci *j*, které jsou na řádcích indexovaných aktuální variací (tyto řádky prochází *for*-cyklus využívající proměnou *i*). Výsledkem je iterátor vracející jedno číslo pro každý sloupec. Přes tento iterátor prochází vnější redukce schovaná ve funkci *product*, která poskytnuté hodnoty pronásobí.

Číslo p je následně přičteno do proměnná *suma* (děje se tak pro každou z $\frac{n!}{(n-k!)}$ variací). I tuto sumaci by bylo lze vyjádřit pomocí redukce (tentokrát s operací sčítání, tato redukce je k dispozici jako standardní funkce *sum*), ale to už by bylo dosti nepřehledné. Na druhou stranu zápis:

```
return sum(product(reduce(logadd, (edg[i, j] for i in v))
                   for j in range(self.size))
           for v in permutations(range(self.size), k)) / fact(k)
```

relativně přesně odpovídá matematickému zápisu (mírně upravený vztah 7 na straně 165):

$$\frac{\sum_{(i_1,i_2,\ldots,i_k)\in V(k)}\left[\prod_{j=0}^{n-1}(a_{i_1 j}\oplus a_{i_2 j}\oplus\ldots\oplus a_{i_k j})\right]}{k!}$$

Sumační funkce nad iterátorem (Pythonský ekvivalent matematické sumy) je použit i v metodě *p_sd*, která vypočítává pravděpodobnost p_{SD}. Parametrem je pravděpodobnost chybového stavu každého jednotlivého modulu p_m. Pro mocnění je použit pythonský standardní operátor „**“ (dvě hvězdičky).

Otázka: Pro implementaci mocnění se běžně používá exponenciální a logaritmická funkce neboť $a^b = e^{b\cdot\ln a}$. Pro přirozené mocnitele ($\in \mathbb{N}$) je však tento algoritmus pomalý a nepřesný. Navrhněte úspornější celočíselný algoritmus a ověřte zda jej Python používá.

5. Kreslíme grafy funkcí

- naučíte se používat knihovnu *matplotlib* pro kreslení grafů funkcí dvou proměnných
- využijete této implementace pro kreslení grafů závislosti pravděpodobnosti p_{SD} na diagnostickém grafu

funkce dvou proměnných, graf funkce

5.1 Implementace

Výpočet a výpis pravděpodobnosti $p_{SD}(DG, p_m)$ pro konkrétní diagnostický graf a konkrétní p_m je snadné:

```
import selfdiagnosis

s = System(System.generateModules(3, PreparatModule, [1]),
        "1-2,3")
print(s.p_sd(0.1))
```

Není to však příliš názorné. Mnohem lepší by bylo nakreslit graf závislosti pravděpodobnosti správné diagnostiky v závislosti na hodnotě p_m (resp. dokonce pro různé diagnostické grafy).

Grafy lze v Pythonu nejsnadněji kreslit pomocí rozhraní *pyplot* z knihovny *matplotlib*. Rozhraní je podobné známému *Matlabu* a většinou si vystačíme s několika příkazy. Nevýhodou tohoto přístupu je téměř nulové znovupoužití kódu a nepříliš flexibilní parametrizace (pro parametrizace lze využít jen globální proměnné).

Proto si vytvoříme několik pomocných tříd, které výrazně usnadní kreslení reálných funkcí jedné proměnné. Speciálním rysem naší implementace bude opět přizpůsobení slabozrakým osobám, tj. jednoduché zvýraznění čar a zvětšení textu.

Nejdříve si připravíme třídu, jejíž instance budou representovat graf dané funkce:

```
class FGraph:
   def __init__(self, callable, label, domain):
      self.callable = callable
      self.label = label
      self.domain = domain
```

Není to zatím nic víc než přepravka uchovávající tři údaje: vlastní funkce v podobě volatelného objektu, popisku funkce (bude použita v legendě) a definiční obor v podobě jediného intervalu (jen na tomto definičním oboru

se bude graf funkce kreslit).

Poté vytvoříme funkci representující graf do něhož můžeme nakreslit libovolný počet funkcí (s jednou soustavou souřadnic).

```
class Plot:
   def __init__(self, title, x_label="", y_label=""):
      self.title = title
      self.xlabel = x_label
      self.ylabel = y_label
      self.funcplots = []

   def __lshift__(self, funcplot):
      self.funcplots.append(funcplot)
```

U grafu lze již v konstruktoru specifikovat titulek a popisky obou os. Navíc lze do grafu přidávat frafy jednotlivých funkcí a to pomocí operátoru „<<“

(zde jsem se inspiroval trochu jazykem C++).

Nyní již lze přistoupit k hlavní metodě, která provádí vlastní kreslení a export do grafického souboru.

```
    def toFile(self, fileName, blackCoefficient=1.0,
                    figSize=(6, 4), dpi=100):
        '''
            vykreslí graf a uloží ho do souboru se jménem
            'fileName'
            blackCofficient: koeficient zvýraznění čar a textu
            (doporučuji 1.00 — 4.00)
            fileSize: šířka a výška obrázku v palcích
            dpi: rozlišení obrázku v bodech na palec
        '''
        # vytvoření prázdného obrázku a jedné dvojice os v něm
        fig = Figure(figSize, dpi, facecolor="white",
```

```
            tight_layout=True)
    ax = fig.add_subplot(1, 1, 1)

    # nastavení vzhledu obrázku
    ax.set_title(self.title)
    ax.grid(lw=blackCoefficient)
    ax.set_xlabel(self.xlabel)
    ax.set_ylabel(self.ylabel)
    ax.axhline(lw=blackCoefficient*1.5, color="k")
    ax.axvline(lw=blackCoefficient*1.5, color="k")

    # nastavení velikosti písma ('fcoef' je koeficient zvětšení)
    fcoef = (1+blackCoefficient/5)

    ax.title.set_fontsize(14*fcoef)
    ax.xaxis.label.set_fontsize(12*fcoef)
```

```
        ax.yaxis.label.set_fontsize(12*fcoef)

        # nastavení zobrazení souřadnic (na ose 'x' i 'y')
        for tick in itertools.chain(ax.xaxis.get_major_ticks(),
                                    ax.yaxis.get_major_ticks()):
            tick.label.set_fontsize(10*fcoef)

        # kreslení grafů jednotlivých funkcí
        for fp in self.funcplots:
            # vytvoření
            xs = np.linspace(fp.domain[0],fp.domain[1], num=100)
            ax.plot(xs, [fp.callable(x) for x in xs],
                    lw=2*blackCoefficient, label=fp.label)

        # přidání legendy
        ax.legend(prop={'size': 10*fcoef})
```

```
        # export do souboru
        canvas = FigureCanvasAgg(fig)
        canvas.print_png(fileName,
                         facecolor=fig.get_facecolor())
```

Protože kód této metody je relativně přehledný, všimněme si detailně jen vlastního kreslení grafu funkce v těle cyklu, jenž iteruje přes všechny grafy funkcí (= instancí třídy *FGraph*).

***Matplotlib* nekreslí přímo funkce, ale jisté body ve 2D prostoru a jejich spojnice. Souřadnice těchto bodu musí být předem spočítány. Nejdříve se vypočte vektor (*NumPy* jednorozměrné pole) x-ových souřadnic. Funkce *numpy.linspace* vrátí *num* (zde tedy 100) rovnoměrně umístěných čísel na intervalu definičního oboru dané funkce (počítaje v to i krajní body intervalu) tj. vektor ($x_0, x_1, \ldots, x_{99}$). Y-souřadnice tvoří druhý vektor, jenž**

je získán aplikací (mapováním) dané funkce na vektor x-ových souřadnic tj. $(f(x_0), f(x_1), \ldots, f(x_{99}))$. Oba vektory jsou pak hlavními parametry kreslící metody *plot*. Plot kreslí postupně body $(x_0, f(x_0))$ až $(x_{99}, f(x_{99}))$ a spojuje je hladkou křivkou.

Otázka: Naše kreslící třída explicitně nedefinuje požadovaný rozsah na osy *y* (obor hodnot funkce) a spokojuje se s automatikou knihovny *matplotlib*. Zjistěte, jak se rozsah stanovuje a v jakých situacích by to mohlo dělat problémy (popřípadě jak tyto situace řešit).

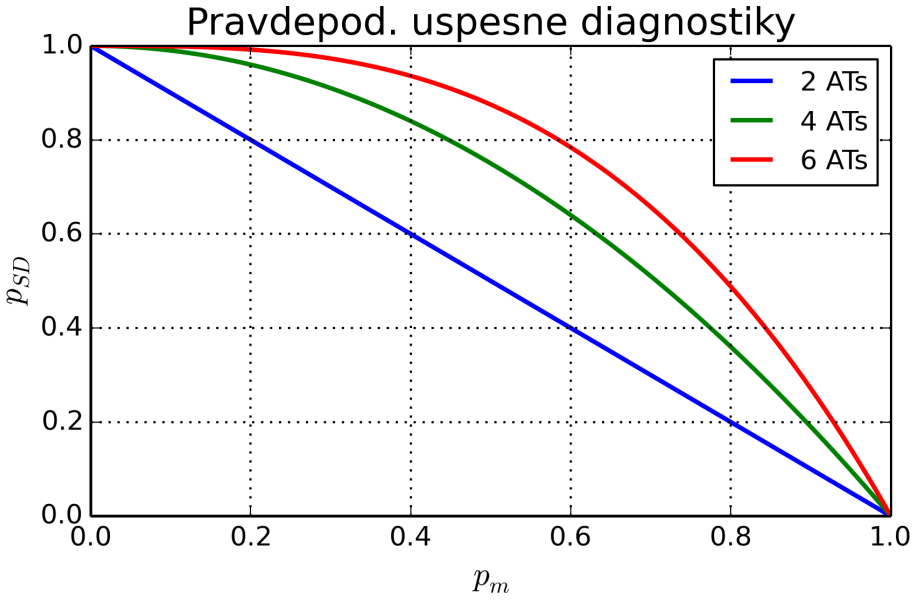
5.2 Příklad použití

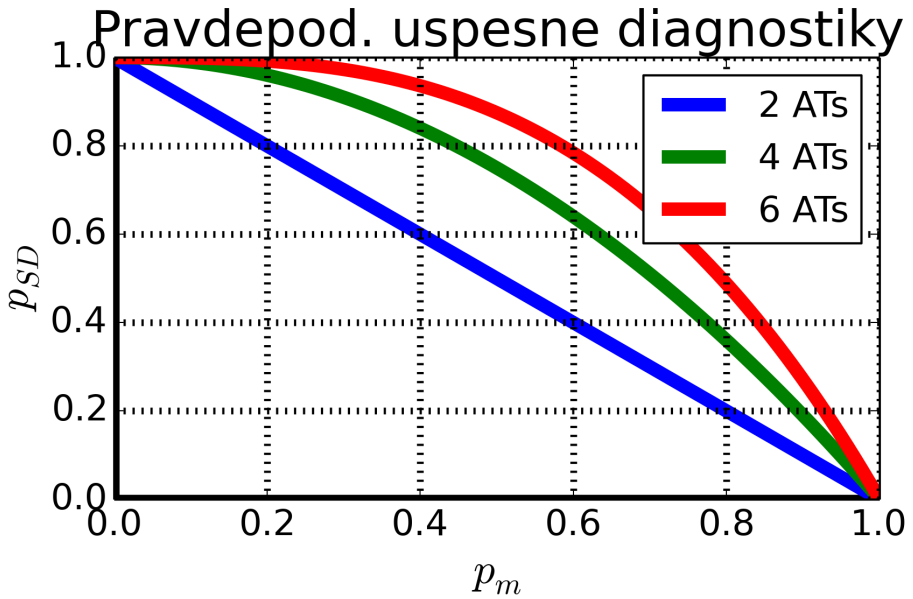
Nově vytvořený modul pro kreslení grafů funkcí (s názvem *graphPlot*) nyní můžeme využít pro vykreslení grafů závislosti pravděpodobnosti p_{SD} na diagnostickém grafu.

```
from selfdiagnosis import System
from graphPlot import Plot, FGraph
```

```
s = System(System.generateModules(3, PreparatModule, [1]),
           "1-2,3") # dvě atomické kontroly
s2 = System(System.generateModules(3, PreparatModule, [1]),
            "1-2,3;2-1,3") # čtyři atomické kontroly
s3= System(System.generateModules(3, PreparatModule, [1]),
           "1-2,3;2-1,3;3-1,2") # šest atomických kontrol

plot = Plot("Pravdepod. uspesne diagnostiky",
         "$p_m$", "$p_{SD}$")
plot << FGraph(s.p_sd, "2 ATs", (0.0, 1.0))
plot << FGraph(s2.p_sd, "4 ATs", (0.0, 1.0))
plot << FGraph(s3.p_sd, "6 ATs", (0.0, 1.0))
plot.toFile("graph.png", blackCoefficient=1.0)
```

Výsledek je vidět na dvou následujících obrázcích. První má ztučňovací koeficient 1.0 (pro ty, kteří mají jiný handicap než slabozrakost), druhý pak 3.0 (pro slabozraké).


Pravdepod. uspesne diagnostiky
2 ATs
4 ATs
6 ATs
p_{SD}
p_m
0.0
0.2
0.4
0.6
0.8
1.0



Při návrhu jakéhokoliv rozhraní je důležité, aby se snadno používal.

Můžeme se pokusit zjednodušit přidávání grafů funkcí. V běžném grafu, totiž funkce sdílejí své definiční obory a je zbytečné je zadávat u každé funkce zvlášť (což může navíc vést k nekonzistencím při úpravách kódu).

Nejdříve učiníme parametr *domain* konstruktoru třídy *FGraph* nepovinným. Stačí do něj přiřadit implicitní hodnotu *None* (která se jistě liší od libovolné n-tice). Ta bude v případě použití (= nezadání třetího parametru) symbolizovat, že funkce převezme implicitní definiční obor z třídy grafu (*Plot*).

```
class FGraph:
  ...
  def __init__(self, callable, label, domain=None):
    ...
```

Proto dodáme parametr *defaultDomain* ke konstruktoru třídy *Plot* s libovolným implicitním intervalem (nikoliv však *None*, vždy musí nějaký definiční obor existovat) a v konstruktoru jej uložíme.

```
class Plot:
  ...
  def __init__(self, title, x_label="", y_label="",
               *, defaultDomain=(-1, 1)):
    ...
      self.defaultDomain = defaultDomain
```

Hvězdička v seznamu parametrů vynutí, že daný parametr bude muset být předán jako pojmenovaný nikoliv jako poziční (metodám a hlavně konstruktorům s velkým počtem pozičních parametrů se vždy vyhýbejte).

Nyní již stačí přepsat metodu *Plot.__lshift__* (operátor levobitového posunu, zde přetížen pro přidávání grafů funkcí):

```
class Plot:
  ...
   def __lshift__(self, funcplot):
      self.funcplots.append(funcplot)
      if funcplot.domain is None: # není—li df. obor definován
         funcplot.domain = self.defaultDomain # použij implicitní
      return self # vrať adresáta metody — podpora řetězení volání
```

Jako malou třešničku na dortu jsme z této metody vrátily odkaz na jejího adresáta, což umožní řetězení operátorů (podobně lze řetězit i běžné metody). Řetězení sice vede k dlouhým špagetózním zápisům, ale někteří programátoři jim dávají přednost.

Nyní můžeme kód vytvářející graf výrazně zkrátit a zpřehlednit.

```
plot = Plot("Pravdepod. uspesne diagnostiky",
         "$p_m$", "$p_{SD}$", defaultDomain=(0.0, 1.0))
# zřetězení dvou oprátorů pro přidání (2 v 1)
plot << FGraph(s.p_sd, "2 ATs") << FGraph(s2.p_sd, "4 ATs")
plot << FGraph(s3.p_sd, "6 ATs")
```

6. Hodnocení diagnostického grafu s ohledem na vlastnosti atomických kontrol

- naučíte se detailnější model diagnostického chování modulů
- naučíte se vypočítat aposteriorní pravděpodobnost bezchybného systému při daním syndromu
- naučíte se měřit časovou efektivitu programů a predikovat dobu vykonávání pro rozsáhlejší vstupy

aposteriorní pravděpodobnost, benchmarking, časová složitost

6.1 Detailnější model atomických kontrol

Všechny výše uvedené vztahy a závěry, vycházeli ze zjednodušujícího předpokladu, že správný modul při atomické kontrole vždy odhalí modul chybný

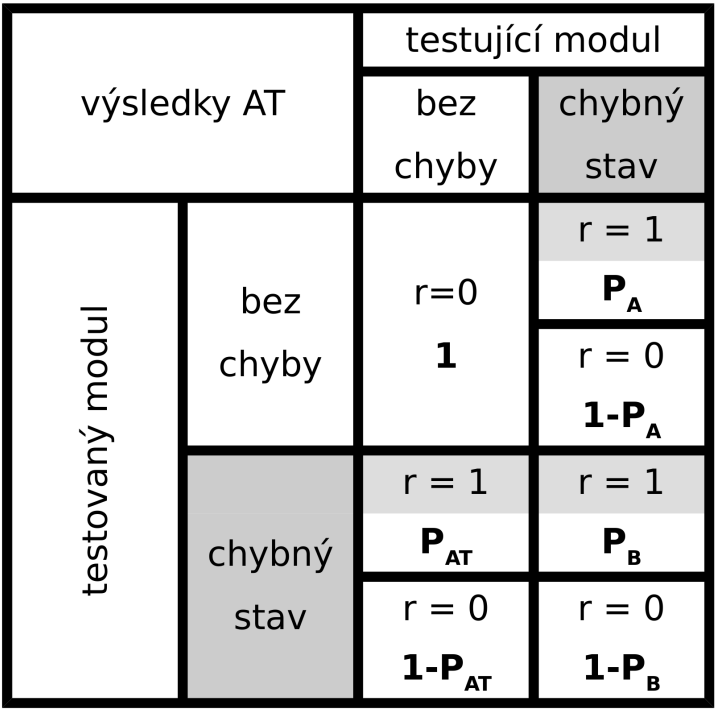
(viz vztah 1 na straně 76). Skutečnost je však jiná, neboť vždy existuje jistá pravděpodobnost, že chyba modulu odhalena nebude. Pravděpodobnost odhalení chybného modulu v rámci atomické kontroly je označována jako **důvěryhodnost** atomické kontroly.

Zohlednění důvěryhodnosti jednotlivých atomických kontrol změní pohled na diagnostický graf a jeho syndrom. Například získání nulového syndromu negarantuje bezchybnost všech modulů.

Z tohoto důvodu je nutné rozhodnout do jaké míry syndrom odráží skutečný stav modulů tj. do jaké míry mu můžeme důvěřovat resp. tuto míru přesně kvantifikovat. To zajišťuje tzv. **aposteriorní pravděpodobnost správnosti systému**, kterou lze přiřadit každému diagnostickému grafu. Pravděpodobnost je ***aposteriorní***, neboť hodnotí systém až po (= a posteriori) provedení atomických kontrol.

Pro výpočet aposteriori pravděpodobnosti vycházíme z dílčích pravděpo-

dobností výsledků atomických kontrol pro jednotlivé stavy systému. Tyto pravděpodobnosti jsou přehledně znázorněny v následující tabulce:


výsledky AT
testující modul
bez chyby
chybný stav
testovaný modul
bez chyby
r=0
1
r = 1
P_A
r = 0
$1-P_A$
chybný stav
r = 1
P_{AT}
r = 1
P_B
r = 0
$1-P_{AT}$
r = 0
$1-P_B$

Pravděpodobnost P_A odpovídá situaci, kdy výsledkem kontroly správného modulu chybným je „1“ (chybný výsledek). P_B je pravděpodobnost výsledku 1 u kontroly chybného modulu chybným modulem (správný výsledek). P_{AT} je pravděpodobnost výsledku, že správný modul zkontroluje chybný modul opět s výsledkem 1 (správný výsledek).

Hodnoty pravděpodobností $P_{A,}, P_B, P_{AT}$ je možno získat buď na základě statistických dat z testů systému (tj. získaných po určité době pozorování systému), nebo z technických specifikací uvedených v dokumentaci systému. Pro účely modelování lze P_A a P_B nastavit cca na 0.5 (odpovídá Preparatově modelu) a P_{AT} s hodnotou blízkou jedné (správný modul ve většině případů správně otestuje chybný).

V našem modelu budeme také (podobně jako v předchozí kapitole) používat další apriorní informaci o systému — pravděpodobnost, že modul je správný označovanou jako P_m. Oproti minulé kapitole bude tato pravděpo-

dobnost vztažena ke každému modulu zvlášť nikoliv k systému jako celku. Navíc je logickým doplňkem pravděpodobnosti p_m (což je pravděpodobnost chybového stavu modulu).

Tento přesnější model nelze modelovat pomocí třídy *PreparatModule* a proto musíme vytvořit novou třídu:

6.2 Implementace

Nová třída bude odvozena z abstraktní třídy *Module* (odvození ze třídy *PreparatModule*, není možné neboť model není speciálním případem Preparatova). Třídu nazveme *PATModule* (podle označení klíčové pravděpodobnosti, nikoliv podle Pata & Mata).

```
class PATModule(Module):
    '''
        Implementace samodiagnostického modulu s apriorní
        pravděpodobností bezchybného stavu (P_m)
```

```
    a detailní specifikací pravděpodbností
    různých výsledků v závislosti na stavu
    testujícího i testovaného modulu.
"""
def __init__(self, ident, initialState,
             *, P_m=0.9, P_at=0.9, P_a=0.5, P_b=0.5):
    super().__init__(ident, initialState)
    assert (initialState == MState.OK and P_m > 0.0 or
            initialState == MState.FAULTY and P_m < 1.0)
    self.P_m = P_m
    self.P_at = P_at
    self.P_a = P_a
    self.P_b = P_b

def atomicControlOn(self, otherModule):
    if self.state == MState.OK:
```

```
            if otherModule.state == MState.OK:
                return 0
            else:
                return 1 if hit(self.P_at) else 0
        else:
            if otherModule.state == MState.OK:
                return 1 if hit(self.P_a) else 0
            else:
                return 1 if hit(self.P_b) else 0
```

Otázka: Jaký je skutečný stav mezi třídou modelující Preparatův a nový detailnější model? Jak by bylo možno pro definici Preparatova modelu využít *lambda* funkci a přiřazení?

Určitý problém přináší spojení dvou úrovní pohledu na modul v jedné třídě. Jeden pohled interpretuje instanci třídy jako popis potenciálního modulu s

určitou pravděpodobností selhání (které navíc bude trvalým stavem), druhý jako konkrétní modul, jehož stav je daný.

Ne zcela jasný je tak vztah mezi parametry *initialState* a *P_m* (pravděpodobnost, že je bezchybný). V našem případě žádný paradox nevzniká, neboť každý z parametrů má jinou funkci (jeden je použit pro inicializaci, druhý bude využit pro výpočet aposteriorní pravděpodobnosti), ale není to zcela košer. Zde je částečně řešen pomocí aserce vylučující kontradikci, když je modul bezchybný a přitom pravděpodobnost jeho bezchybnosti je 0 a vice versa. To není dokonalé, což se projeví v kapitole 8.3 na straně 323.

6.3 Aposteriorní pravděpodobnost správnosti systému

Pokud známe apriorní pravděpodobnosti správnosti u všech modulů P_{m_i} (kde i je index modulu), lze snadno vypočítat i apriorní pravděpodobnost bezchybnosti systému (tj. pravděpodobnost, že všechny moduly jsou správné):

$$\prod_{i=0}^{n-1} P_{m_i}$$

Po provedení atomických kontrol a po obdržení syndromu však můžeme vypočítat aposteriorní pravděpodobnost správnosti systému (tj. správnosti všech jeho modulů), která se od apriorní pravděpodobnosti liší (získali jsme dodatečnou informaci).

Výpočet aposteriorní pravděpodobnosti začneme definicí všech hypotéz o možném stavu systému (stav systému uspořádanou množinou stavů jeho modulů).

Pro systém s dvěma moduly M_1 a M_2 jsou to tyto čtyři hypotézy:

H_0: M_1 **je správný** M_2 **je správný**

H_1: M_1 **je správný** M_2 **je chybný**

H_2: M_1 **je chybný** M_2 **je správný**

H_3: M_1 **je chybný** M_2 **je chybný**

Pro systém s více moduly bude těchto hypotéz o trochu málo více (to byla ironie, ve skutečnosti roste počet hypotéz exponenciálně, neboť je zřejmé, že je roven 2^n).

Poté spočítáme apriorní pravděpodobnosti všech hypotéz:

$$P(H_0) = P_{m_1} \cdot P_{m_2}$$

$$P(H_1) = P_m \cdot (1 - P_{m_2})$$

$$P(H_2) = (1 - P_{m_1}) \cdot P_{m_2}$$

$$P(H_3) = (1 - P_{m_1}) \cdot (1 - P_{m_2})$$

Poté vyčíslíme podmíněné pravděpodobnosti pro každou hypotézu a daný syndrom. Podmíněné pravděpodobnosti $P(R/H_i)$ jsou pravděpodobnosti získání syndromu R po provedení atomických kontrol za situace, kdy stav systému odpovídá hypotéze H_i. Tyto pravděpodobnosti získáme jako součin pravděpodobností výsledků jednotlivých atomických kontrol při dané hypotéze $P(R_{AT}/H_i)$.

Poté lze vypočítat aposteriorní pravděpodobnost hypotézy H_i při syndromu R pomocí následujícího bayesovského vztahu:

$$P(H_i/R) = \frac{P(H_i) \cdot P(R/H_i)}{\sum_{j=0}^{2^n-1} (P(H_j) \cdot P(R/H_j))} \qquad \textbf{(8)}$$

Nás bude zajímat především pravděpodobnost stavu, kdy jsou všechny moduly správné tj. H_0, ale lze samozřejmě vypočítat i pravděpodobnosti ostat-

ních hypotéz.

6.4 Implementace výpočtu aposteriorní pravděpodobnosti

Než začneme implementovat algoritmus pro výpočet aposteriorní pravděpodobnosti, je nutné si ozřejmit jeho výstup a především vstupů. Algoritmus počítá jedinou skalární hodnotu (pravděpodobnost) ze dvou relativně komplexních vstupů:

1. množiny modulů a jejich očekávaných stavů (= kombinace stavů testované hypotézy) tj. hypotéza
2. syndromu získaného pomocí atomických kontrol nad systémem. Stavy modulů v tomto systému nemusí odpovídat hypotéze (lze testovat i chybnou hypotézu).

Volání metody může mít potenciálně dvě různé formy (podle toho, jaký objekt bude považován za adresáta)

```
hypotéza.apost_probability(syndrom)
```

nebo

```
syndrom.apost_probability(hypotéza)
```

Hypotetický systém může být (zdánlivě) representována pomocí instance třídy *System* (s dodanou metodou *apostProbability*). Instance této třídy sice obsahují potřebné informace (dokonce mnohem více informací), ale nelze ji použít. Stačí použít důkaz sporem.

Stačí ověřit dvě možnosti:

- **hypotéza je reprezentována stejným systémem jako je systém syndromu. To však nejde, neboť hypotéza může mít jiné stavy modu-**

lů než systém syndromu (a to po celou dobu vykonávaní metody). Především z tohoto důvodu jsou moduly i systém navrhovány jako neměnné (po vytvoření by neměly být modifikovány a my jsme tak nikdy neučinili). Obecně není vhodné odkazovat z více (zároveň použitelných) proměnných měnitelné objekty

- **hypotéza je representována jiným systémem (jinou instancí třídy *System*). Zde je problém udržení konzistence. Jak snadno dosáhneme toho, že obě instance mají diagnostický graf. Nestačí jen stejný počet, stejné musí být i objekty modulů, neboť na ty definují identitu. Navíc i moduly nesou informace, jaké potom použijeme, ty z instance hypotézy nebo ty ze systému na než je vázán syndrom (ten svůj systém odkazuje).**

Pro hypotézu proto vytvoříme novou třídu, která bude inicializována pouze počtem modulů, a seznam pozic chybných stavů (snadné zadání hypotézy

H_0 prázdným seznamem). Navíc bude implementovat metodu *apostProbability* (volíme tedy první zápis volání, v němž je adresátem hypotéza a nikoliv syndrom). Interní representace hypotézy však bude jiná (viz níže).

Implementaci zahájíme rozšířením rozhraní tříd modulů o novou metodu, která pro daný výsledek atomické kontroly vrátí jeho pravděpodobnost. Metodu lze implementovat pro všechny zatím zavedené modely atomických kontrol, tj. můžeme ji zahrnout již do rozhraní abstraktní třídy Module, i když samozřejmě bez konkrétní implementace (třída *Module*, žádný konkrétní model atomických kontrol neimplementuje).

```
class Module:
  ...
  def probabilityOfResult(self, otherModule, result,
                          selfState=None, otherState=None):
      '''
          vrací pravděpodobnost výsledku 'result'
```

```
        u atomické kontroly modulu
        'otherModule' modulem 'self'.

        Pomocí atributů 'selfState' a 'otherState'
        lze popřípadě určit jiný než aktuální stav modulu.
        (pokud je hodnota parametru různá od 'None')
    '''
    raise NotImplementedError("abstract method")
```

Metoda má dva povinné parametry (nepočítaje *self*). *OtherModule* je (stejně jako u atomické kontroly) testovaný modul, *result* je výsledek, jehož pravděpodobnost nás zajímá. Při testování aposteriorní pravděpodobnosti nás však nezajímá pravděpodobnost výsledku při skutečném stavu modelů, ale při stavu hypotetickém (např. u hypotézy H_0 budou oba moduly správné). Proto jsou dodány dva parametry, pomocí nichž lze předat obecně jiné

(tj. zde hypotetické) stavy modulů (jsou-li zadány, nahradí reálné stavy).

Implementace pro objekty třídy *PATModule* je obdobou implementace modelu atomické kontroly (obě metody jsou do jisté míry duální). Nejdříve si připravíme dvě pomocné metody:

```
# pomocné funkce
def resultProbability(p_of_one, result):
    '''
        vrací pravděpodobnost výsledku 'result',
        je—li pravděpodobnost výsledku '1' rovna 'p'
    '''
    return p_of_one if result == 1 else 1-p_of_one


def coalesce(*args):
    '''
```

```
    vrací první neprázdný parametr (tj. různý od None)
  '''
  for arg in args:
     if arg is not None:
        return arg
  assert False, "All parameters are None"
```

První zjednodušuje vrácení pravděpodobnosti pro oba možné výsledky. Je-li výsledek 1 pak vrací předanou pravděpodobnost, u výsledku 0 pak doplněk této pravděpodobnosti ($1 - p$).

Zatímco první metoda je velmi specializovaná (je vázána na námi použitou strukturu modelů AT), je metoda *coalesce* zcela univerzální a v praxi ji můžete použít v mnoha pythonských programech (ve skutečnosti je spíše překvapivé, že není vestavěna). Metoda očekává dva a více parametrů a vrací první z nich, který není *None* (tj. první se skutečnou hodnotou). Po-

užití s jedním či dokonce žádným parametrem je syntakticky možné (pole parametrů *args*, označené prefixem '*' bude prázdné či jednoprvkové), ale zcela neužitečné.

```
class PATModule:
  ...
  def probabilityOfResult(self, otherModule, result,
                          selfState=None, otherState=None):
      selfState = coalesce(selfState, self.state)
      otherState = coalesce(otherState, otherModule.state)
      if selfState == MState.OK:
         if otherState == MState.OK:
            return resultProbability(0.0, result)
         else:
            return resultProbability(self.P_at, result)
      else:
```

```
        if otherState == MState.OK:
            return resultProbability(self.P_a, result)
        else:
            return resultProbability(self.P_b, result)
```

Z důvodů úplnosti je tato metoda implementována i u třídy *PreparatModule*.

```
    def probabilityOfResult(self, otherModule, result,
                            selfState=None, otherState=None):
        selfState = coalesce(selfState, self.state)
        otherState = coalesce(otherState, otherModule.state)
        if selfState == MState.OK:
            return resultProbability(
                0.0 if otherState == MState.OK else 1.0, result)
        else:
            return resultProbability(self.p, result)
```

Po této přípravě je již možné uvést třídu, jejíž instance representují hypotézy:

```
class Hypothesis:
   def __init__(self, size, value):
      self.size = size
      self.max = 2**size
      self.value = value

   def __getitem__(self, i):
      '''
         vrací stav i—tého modulu v hypotéze (jako 0,1)
      '''
      return (self.value >> i) & 1

   @staticmethod
```

```
    def fromFaultyModules(size, faultyModuleIndexes=[]):
        '''
            tovární metoda vytvářející hypotézu na základě
            seznamu indexů chybných modulů
        '''
        value = 0
        for index in faultyModuleIndexes:
            value |= (1 << index)
        return Hypothesis(size, value)

    @staticmethod
    def h0(size):
        '''
            tovární metoda vytvářející hypotézu H_0
            (všechny moduly jsou správné)
        '''
```

```
        return Hypothesis(size, 0)

    def nextHypothesis(self):
        '''
            vrací nový objekt hypotézy H_n+1
        '''
        return Hypothesis(self.size,
                          (self.value + 1) % self.max)

    def __eq__(self, other):
        return (self.value == other.value
                and self.size == other.size)

    def __ne__(self, other):
        return (self.value != other.value
                or self.size != other.size)
```

```
def genAllHypotheses(self):
    '''
        iterátor generující posloupnost všech možných hypotéz
        počínaje hypotézou 'self'.
    '''
    yield self
    nh = self.nextHypothesis()
    while nh != self:
        yield nh
        nh = nh.nextHypothesis()

def __str__(self):
    return bin(self.value)
```

Pokud známe počet modulů n, lze hypotézy generovat jako n-prvkové variace s opakováním (záleží na pořadí) z množiny {0,1}. Pro $n = 2$ jsou to tyto variace $\{(0,0),(0,1),(1,0),(1,1)\}$ (0 symbolizuje správný modul a 1 chybný). Není potřeba příliš hluboké úvahy abychom zjistili, že existuje 2^n těchto variací a že je lze všechny representovat pomocí čísel $0 \ldots 2^n - 1$, kde n je počet modulů v příslušném systému. Stačí si představit binární zápis čísel s n binárními číslicemi od $0 \ldots 0$ (samé nuly) do$1 \ldots 1$ (samé jedničky).

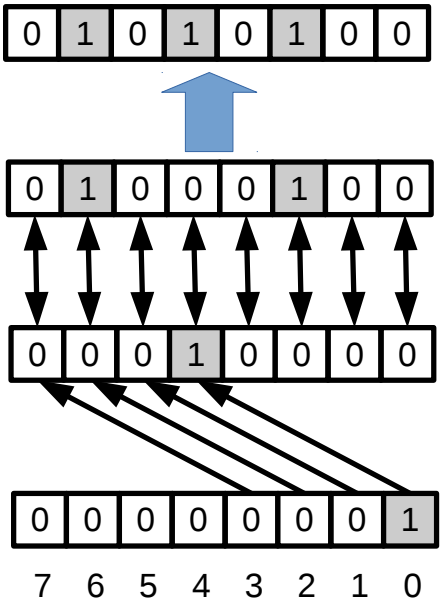
Každý stav modulu je tak v hypotéze representován jen jedním bitem. To přináší zcela minimální požadavky na paměť a velmi rychlé přístupy (rychlejší než v případě polí). Urychlení se však ve výsledku neprojeví, neboť nikoliv operace s hypotézou, ale operace s diagnostickým grafem zaujímají podstatnou část doby vykonávání těla vnějšího cyklu (přes iterace). Navíc počet iterací roste exponenciálně tj. i časová složitost je exponenciální a (malé) lineární zrychlení se téměř neprojeví (je jedno zda bude algoritmus

pro 40 uzlů běžet 1000 let nebo 820 let).

Primárním prostředkem tvorby instancí hypotéz je konstruktor, který očekává velikost hypotézy (= počet modulů) a číslo representující stavy. Pro snadnější použití je implementována i tovární metoda pro vytvoření libovolné hypotézy ze seznamu chybných modulů (moduly jsou v čísle indexovány od nejnižšího bitu, tj. v binárním zápise zleva doprava). K dispozici je i metoda pro snadné vytváření základní hypotézy H_0 (všechny moduly správné).

Při nastavování jednotlivých bitů musí být použity tzv. bitové operátory, které provádějí základní logické operace mezi jednotlivými bity celého čísla. My potřebujeme nastavit i-tý bit což lze provést pomocí bitového operátoru „|“ (bitové OR). Nejdříve vezmeme jedničku a posuneme ji o i-bitů doleva (operace '<<'). Tím získáme tzv. masku, která obsahuje samé nuly, jen na i-té pozici je jedničkový bit. Pak vezmeme původní hodnotu čísla a aplikujeme

na něj masku pomocí bitového or. Bity ležící oproti nulám se nezmění, bit na i-té pozici se nastaví na 1 (resp. zůstane jedničkový). Pak se nová hodnota zapíše do původní proměnné (zápis x|=y je zkratka za x = x | y, stejně jako ve všech jazycích ovlivněných C)



(1 << 4) | 0x44 = 0x54 = 84
bit 4 nastaven

původní hodnota 0x44 = 68
bit 4 = 0

maska: 1 << 4 = 0x10 = 16

posunutí o 4 bity vlevo

1

indexy bitů

Pro přístup k jednotlivým bitům se používá operátor indexace, který je implementován opět pomocí bitových operací. Číslo je nejdříve posunuto o hodnotu indexu vpravo, tak že se hledaný bit objeví na pozici s indexem 0 (původní číslo se nemění, čísla jsou v Pythonu vždy neměnná, jen se vrátí nové posunuté číslo). Na nové číslo je aplikován operátor „&“ (bitové AND) s operandem 0, který zajistí vynulování všech bitů kromě toho s indexem 0. Metody tak vrátí 0 nebo 1 podle původního obsahu i-této bitu.

Jádrem třídy rozhraní třídy hypotéz je generátor, který vytváří iterátor poskytující všechny hypotézy počínaje hypotézou, nad níž je volán (tj. všech 2^n hypotéz). Interně používá funkci, která poskytuje následující iterátor. Díky representaci je to otázka přičtení jedničky (výsledek je ještě převeden do rozsahu $0 \ldots 2^n - 1$ operací zbytek po dělení, aby nedošlo k přetečení). Při implementaci iterátoru je klíčový jeho lenivý charakter (zde je lenost ctností). Vytvoření seznamu všech hypotéz by bylo již pro relativně malé

systémy extrémně paměťově náročné (systém 20 modulů by vyžadoval cca 20MiB).

Poslední implementovanou metodou třídy *Hypothesis* je cílová metoda *apostProbability*:

```
class Hypothesis:
  ...
   def apost_probability(self, syndrome):
      '''
          výpočet aposteriorní pravděpodobnosti hypotézy 'self'
          (o stavu modulů) při daném syndromu.
      '''
      assert self.size == syndrome.system.size
      s = syndrome.system
      m = s.moduleList
      suma = 0.0
```

```
nom = None
# cyklus přes všechny hypotézy počínaje H_O
for h in Hypothesis.h0(self.size).genAllHypotheses():
    # výpočet apriorní pravděpodobnosti hypotézy P(H)
    # jako součinu pravděpodobností stavů všech modulů
    ph = Π(m[i].P_m if h[i] == 0 else 1-m[i].P_m
         for i in range(self.size))
    # výpočet pravděpodobnosti P(R/H) syndromu
    # pro stav daný hypotézou jako součinu pravděpodobnosti
    # dílčích atomických kontrol
    prh = Π(m_i.probabilityOfResult(m_j, r,
                    selfState=MState(h[s.mpos(m_i)]),
                    otherState=MState(h[s.mpos(m_j)]))
          for (m_i, m_j, r) in syndrome)
    # přičtení do sumy ve jmenovateli bayesovského vztahu
    suma += ph * prh
```

```
            if h == self:
                # čital beyesovského vztahu (pro hypotézu self)
                nom = ph * prh
        return nom/suma
```

Tato třída je relativně přesným přepisem bayesovského vztahu. Pro zkrácení zápisu je pro funkci multiplikativního produktu použit identifikátor Π namísto delšího *product* (tak jsme naši funkci pojmenovali v kapitole 4.2 na straně 168). Python podporuje i identifikátory z exotičtějších abeced a snadné je i vytváření aliasů funkcí. Stačí na nejvyšší úrovni programu provést přiřazení:

```
Π
 = product
```

Výpočet pravděpodobnosti syndromu využívá, metodu *probabilityOfResult*, již jsme dodali k definici každé třídy odvozené ze třídy *Modul*. Zápis komplikuje jen nutnost získaní indexu příslušných modulů v daném systému (*mpost*) a získání jeho hypotetického stavu pomocí indexace. V každé iteraci cyklu se hodnota $P(H) \times P(R/H)$ přičte k sumě, která se stane jmenovatelem, a je-li aktuálně iterovaná hypotéza rovna té, jejíž pravděpodobnost zjišťujeme (= *self*) je tato hodnota uložena i do proměnné *nom* (čitatel bayesovského vzorce). Po skončení cyklu tak stačí tyto proměnné vydělit.

Na závěr se ještě podíváme jak se nově vytvořený kód použije na jednoduchý graf s dvěma správnými moduly a jednou atomickou kontrolou (vracející výsledek 0)

```
m = System.generateModules(2,
     lambda name, initialState:
        PATModule(name, initialState,
                P_at=0.8, P_a=0.5, P_b=0.5, P_m=0.9),
```

```
    [])
s = System(m, "1-2") # vytvoření systému
p = round(Hypothesis.h0(2).apost_probability(s.getSyndrome()),
        2)
print(p)
```

Výsledek by měl být 0.93 (=94%). Díky dodatečné informaci z atomické kontroly se pravděpodobnost zvýšila z původních 81% (apriorní pravděpodobnost při P_m=0,9 o 12%. Vestavěná funkce *round* slouží k zaokrouhlování čísel (zde na dvě desetinná čísla).

Otázka: Zkuste zobrazit graf funkce $P(H_0/R)$ v závislosti na hodnotě P_m a P_{at} (s grafy více funkcí jedné proměnné).

6.5 Benchmarking

Algoritmus pro výpočet aposteriorní pravděpodobnosti má zcela jistě exponenciální časovou složitost $O(\mathrm{e}^x)$, neboť jádro algoritmu se provádí v cyklu, který se provádí 2^n krát (kde n je počet modulů).

Ve většině „dobrých“ publikací o programování a algoritmizaci, se dozvíte, že exponenciální časová složitost prakticky brání použití algoritmu pro větší hodnoty n (obecně již v malých desítkách). Proto je vhodnější se těmto algoritmům vyhýbat a to buď použitím algoritmus s nižší časovou složitostí (polynomiální) nebo pokud takovýto algoritmus neexistuje (tzv. NP-úlohy), pak použitím heuristického přístupu (poskytuje suboptimální řešení, které je však v praxi většinou vyhovující).

Je však situace opravdu tak špatná? Vždyť pro námi testované grafy s počtem modulů menším než pět, je výsledek vrácen bez viditelného prodlení.

Nic nám nebrání otestovat rychlost provádění algoritmu aposteriorní prav-

děpodobnosti pro různě velké grafy (tzv. benchmarking) a na základě aproximace určit jeho chování i pro větší n (kde už by čekání na dokončení bylo nepraktické, nebo dokonce nemožné).

Dříve než začneme měřit čas, musíme si připravit pomocný nástroj na generování „rozumných“ diagnostických grafů, neboť ruční vytváření většího počtu středně velkých grafů je nepraktické. Dobrým východiskem jsou optimální diagnostické grafy, které jsou t_{max}-diagnostikovatelné a mají pro tento účel minimální počet atomických kontrol $\frac{n(n-1)}{2}$ (počet kontrol roste kvadraticky, tj. i pro relativně malé grafy počet hran znemožňuje ruční zadání).

V prvé řadě je nutné změnit konstruktor instancí třídy *System*. Ten očekává hrany v podobě relativně kompaktních řetězcových popisovačů, což je vhodné pro lidi, méně již pro počítače. Kód pro generování optimálních grafů, by byl nucen generované dvojice modulů *(testující, testovaný)* převádět na ře-

těžce, které by následně byly parserem převáděny opět na dvojice indexů. Jednodušší je předat přímo seznam či iterátor dvojic.

Python bohužel nemůže podporovat přetížené verze metod a tím i konstruktorů, které se liší jen typem parametrů (Python je staticky typován a typy parametrů se nikde neuvádějí). Naštěstí, lze tento typ polymorfismu emulovat větvením na základě běhové identifikace typů. Je to běžný pythonský idiom, neměl by však být používán příliš často, neboť znepřehledňuje program (většinou je lepší využití různě pojmenovaných továrních metod).

```
class System:
  ...
  def __init__(self, modules=None, graph=""):
      """
          vytváří systém s danými moduly a diagnostickým grafem.
          Pořadí modulů je pro některé operace signifikatní (např.
          při vytváření matice sousednosti).
```

```
        Parametr graph slouží k zadání atomických kontrol (tj.
        hran diagnostického grafu). Hrany mohou být zadány
        iterátorem poskytujícím dvojic
        (index_testujícího_modul, index_testovaného_modulu),
        nebo stručnějším řetezcovou representací.
        :type modules: list
    '''
    self.moduleList = list(modules)
    self.modules = {m.id: m for m in self.moduleList}
    self.size = len(self.modules)
    self.dg = set()
    # zde se konstruktor větví
    if isinstance(graph, str): # 'graph' je řetězec
        self._processGraphInput(graph)
    else: # je to něco jiného (seznam dvojic)
```

```
            self.dg = {
              AtomicCheck(m_i=self.moduleList[i],
                    m_j=self.moduleList[j]) for i, j in graph}
```

Nový je konec konstruktoru, kde se na základě typu objektu v proměnné *graph*, větví vykonávání na dvě různé inicializace . Používá se přitom vestavěná metoda *isinstance*, která umožní za detekovat typ (třídu) proměnné. Inicializace pomocí seznamu dvojic (větev *else*) využívá množinovou komprehenzi (vytváří množinu na základě iterátoru).

Nyní již lze implementovat metodu pro generování seznamu hran v podobě seznamu dvojic indexů. Statická metoda s velmi dlouhým názvem *System.optimalSampleATList* je ve skutečnosti generátorem iterátoru (tj. vrací iterátor poskytující postupně jednotlivé dvojice indexů):

```
class System:
 ...
 @staticmethod
  def optimalSampleATList(n, t=None):
     assert n > 2
     t = coalesce(t, (n-1)//2)
     assert t <= (n-1)//2
     for i in range(n):
        for j in range(1, t+1):
           yield (i, (i+j) % n)
```

Pro každé n existuje více optimálních grafů, kód generuje jen jeden z nich. Z každého modulu vychází t_{max} atomických kontrol do t_{max} modulů, které leží v pořadí za ním se zacyklením (tj. za modulem s nejvyšším indexem se nachází modul první, druhý atd.) Je zřejmé že diagnostický graf obsahuje

$t_{max} \cdot n = \frac{n(n-1)}{2}$ **atomických kontrol.**

Program pro benchmarking počítá čas provedení algoritmu kontroly pro optimální grafy s různými počty modulů a zapisuje je do textového souboru v jednoduchém formátu (počet modulů, tabulátor, čas). Teprve tato data jsou následně zpracována (generování může trvat i celé hodiny a je tudíž nemyslitelné, že by se provádělo při každém zpracování či zobrazení).

```
from selfdiagnosis import System, PATModule
from hypothesis import Hypothesis
from contextlib import contextmanager
from time import process_time
from sys import stderr



@contextmanager
def benchmark(taskIndex, outTextStream, *, timeout=1e10):
```

```
    '''
        vytváří context managet zapisující při výstupu
        do textového proudu 'outTextStream'
        index úlohy 'taskIndex' typicky číslo
        a čas provedení úlohy ve vteřinách.

        Pokud je doba provedení větší než 'timeout'
        vteřin, je po zapsání vyvolána výjimka
        TimeoutError.
    '''
    startTime = process_time()
    yield
    endTime = process_time()
    elapsed = endTime - startTime
    print("{0}\t{1:.6f}".format(i, elapsed),
          file=outTextStream)
```

```
        outTextStream.flush()
        if elapsed > timeout:
            raise TimeoutError

with open("/tmp/sd-benchmark.txt", "wt") as output:
    for i in range(3, 100):
        m = System.generateModules(i,
            lambda name, initialState:
                PATModule(name, initialState,
                    P_at=0.9, P_a=0.5, P_b=0.5, P_m=0.9),
            [])
        g = System.optimalSampleATList(i)
        s = System(m, g)
        with benchmark(i, output, timeout=1000.0):
            Hypothesis.h0(i).apost_probability(s.getSyndrome())
```

Pro snazší a přehlednější zápis časoměrného kódu je využita podpora tzv. správců kontextu (*context manager*) v Pythonu ve spolupráci s příkazem *with*.

Běžnějším a standardnějším druhem správce kontextu je správce kontextu otevřeného souboru. V příkazu *with* na začátku hlavního programu se volá metody *__entry__* nad objektem vzniklým voláním funkce *open* (je to otevřený textový proud). Objekt se přiřadí do proměnné *output* (konstrukcí *as* proměnná). Navíc se nad ním zavolá metoda *__entry__* (ta však u souborů nemusí nic provádět).

Od této chvíle je objekt proudu k dispozici uvnitř těla příkazu *with*. Po opuštění těla příkazu *with* a to jakýmkoliv způsobem (dosažením konce, vyhozením nezachycené výjimky, výskokem pomocí příkazu *return*.) se zavolá nad objektem proudu metoda *__exit__*. Ten v tomto případě uzavře příslušný proud a otevřený soubor.

Toto použití příkazu *with* je příkladem tzv. správce prostředků. Prostředek je bezpečně uvolněn hned jak se přestane využívat a to vždy (i když vznikne výjimka či předčasně vyskočíme těla příkazem *return,* apod.)

Po otevření logovacího souboru (se jménem */tmp/sd-benchmark.txt*, je otevřen pro textový zápis = „wt"), program vstoupí do cyklu, v němž postupně vytváříme systémy se 3,4,... moduly (číslo 100 je zde de facto nekonečnem).

Vnořený příkaz *with* již přímo souvisí s měřením času a logováním. Metoda benchmark (jejíž definice je na začátku programu) vytvoří objekt, u něhož se stejně jako v případě souboru nejdříve zavolá metoda *__entry__*. V této metodě se zapamatuje startovní čas. Pak se zavolá tělo s výpočtem aposteriorní pravděpodobnosti (ta se nikam nevypisuje ani neukládá). Poté se zavolá metoda *__exit__* souboru vzniknuvšího v příkazu *with,* neboť program opouští jeho tělo. V této metodě se zaznamená koncový čas, odečte

se od něj čas startovní a rozdíl se spolu s číslem úlohy (zde je to počet modulů) zapíše na konec logovacího souboru (otevřený soubor a číslo úlohy jsou parametry metody *benchmark*, která tento objekt vytvořila).

Pro vytvoření tohoto objektu stačí definovat třídu, s konstruktorem a implementací metody *__enter__* a *__exit__*. Objekty takovéto třídy by splňovaly protokol správce kontextu a bylo by je možno využívat v příkazu *with* (ve skutečnosti je tento protokol obecnější, ale pro většinu použití tento popis stačí)

Existuje však i jednodušší zápis. Stačí použít korutinu (= generátor iterátoru) opatřenou dekorátorem *contextlib.contextmanager* (*contexlib* je standardní modul). První část korutiny definuje co se má provést při vstupu do kontextu (u příkazu *with*). Pak se korutina dočasně opustí (příkaz *yield*) a vykoná se tělo příkazu *with* (u nás tedy výpočet pravděpodobnosti). Pak se řízení vrátí do korutiny a ta vykoná funkci metody *__exit__*. Vypočítá rozdíl

času (bez problémů vidí lokální proměnnou *startTime*) a vypíše je do souboru. Pro jistotu ještě zajistí vyprázdnění aplikačního bufferu, tj. text se vypíše bezprostředně (přesněji téměř bezprostředně, ještě jsou zde vyrovnávací paměti operačního systému a disku).

Úkol: Pro zjištění času provedení kódu, jsem využil funkci *time.process_time()*. Jsou nějaké další alternativy? Jaká je jejich výhoda?

Pro snadnější použití jsem přidal ještě parametr *timeout*. Je-li poslední rozdíl času větší než předaný *timeout*, pak je vyvolána výjimka *TimeoutError*, která může program ukončit (může však samozřejmě být i zachycena). Timeout využíváme i zde — benchmark ukončen, pokud doba trvání posledního výpočtu překročí 1000 sekund.

Po spuštění programu a jeho asi 40 minutovém běhu se na mém počítači vytvořil soubor s tímto obsahem:

```
3 0.000407
4 0.000927
5 0.004073
6 0.009659
7 0.031955
8 0.072272
9 0.213974
10 0.950450
11 2.573555
12 4.284591
13 7.722467
14 16.317449
15 39.467869
16 84.246028
```

```
17 205.168397
18 435.350886
19 1045.045866
```

I z tohoto výpisu je zřejmé, že algoritmus má exponenciální složitost, neboť čas výpočtu pro systém s $n + 1$ uzly se více než dvakrát zvyšuje.

Ještě více informací získáme, pokud funkci závislosti času výpočtu aproximujeme metodou nejmenších čtverců a graficky znázorníme.

Pro zobrazení grafu funkce můžeme použít třídu ***Plot***, kterou jsem vytvářeli v kapitole 5 na straně 180. Před použitím si ji však trochu vylepšíme. Kromě vizuálního vyladění (méně obtěžující a lépe vypadající mřížka a částečně průhledná legenda) přidáme podporu logaritmické osy (což je v případě zobrazení exponenciály nezbytnost) a především tzv. pravítek tj. horizontálních a vertikálních úseček s popisy. Tento mechanismus (zne)užijeme i

pro kreslení jednotlivých bodů (zde pro zobrazení prokládaných dat).

Třída pravítek je relativně jednoduchá, je to jen přepravka uchovávající informace o pozici pravítka a jeho případném popisku). Typ pravítka je určen tím, jaké souřadnice jsou zadány.

```
class Ruler:
   def __init__(self, x=None, y=None, label=""):
      self.x = x
      self.y = y
      self.label = label

   @property
   def type(self):
      if self.x is None:
         return "x-axis"
      if self.y is None:
```

```
            return "y-axis"
        if self.x is not None and self.y is not None:
            return "point"
        raise NotImplemented("Unposit.rulers are unimplemented")
```

O něco složitější je kreslení, které se provádí ve třídě *Plot* (v pomocné metodě *_plotRuler*).

```
class Plot:
    def __init__(self, title, x_label="", y_label="",
                 *, defaultDomain=(-1, 1), logScale=False):
        self.title = title
        self.xlabel = x_label
        self.ylabel = y_label
        self.funcplots = []
        self.defaultDomain = defaultDomain
        # nové, řídí zda je osy 'y' logaritmická
```

```
        self.logScale = logScale
        # seznam pravítek
        self.rulers = []

    def __lshift__(self, gobject):
        '''
            přidání grafického objektu
            prozatím jsou podporovány grafy funkcí a pravítka
        '''
        # dynamická implementace přetížení
        if isinstance(gobject, FGraph): # přidání funkce
            self.funcplots.append(gobject)
            if gobject.domain is None:
                gobject.domain = self.defaultDomain
        if isinstance(gobject, Ruler): # přidání pravítka
            self.rulers.append(gobject)
```

```
        return self

    def toFile(self, fileName, blackCoefficient=1.0,
                    figSize=(6, 4), dpi=100):
        '''
            vykreslí graf a uloží ho do souboru se jménem
            'fileName'
            blackCofficient: koeficient zvýraznění čar a textu
            (doporučuji 1.00 — 4.00)
            fileSize: šířka a výška obrázku v palcích
            dpi: rozlišení obrázku v bodech na palec
        '''
        # vytvoření prázdného obrázku a jedné dvojice os v něm
        fig = Figure(figSize, dpi, facecolor="white",
                    tight_layout=True)
        ax = fig.add_subplot(1, 1, 1)
```

```
# nastavení logaritmické osy 'y'
if self.logScale:
   ax.set_yscale("log", basey=10.0)

ax.set_title(self.title)
ax.set_xlabel(self.xlabel)
ax.set_ylabel(self.ylabel)
# osu 'x' nelze v případě logaritmické osy 'y' zobrazit
if not self.logScale:
   ax.axhline(y=0, lw=blackCoefficient*1.5, color="k")
ax.axvline(lw=blackCoefficient*1.5, color="k")

# nastavení velikosti písma ('fcoef' je koeficient zvětšení)
fcoef = (1+blackCoefficient/5)
```

```
ax.title.set_fontsize(14*fcoef)
ax.xaxis.label.set_fontsize(12*fcoef)
ax.yaxis.label.set_fontsize(12*fcoef)

# nastavení zobrazení souřadnic (na ose 'x' i 'y')
for tick in itertools.chain(ax.xaxis.get_major_ticks(),
                            ax.yaxis.get_major_ticks()):
      tick.label.set_fontsize(10*fcoef)

# kreslení grafů jednotlivých funkcí
for fp in self.funcplots:
   # vytvoření
   xs = np.linspace(fp.domain[0], fp.domain[1], 100)
   ax.plot(xs, [fp.callable(x) for x in xs],
         lw=2*blackCoefficient, label=fp.label)
```

```
# cyklus pro kreslení pravítek (nově přidáno)
for ruler in self.rulers:
   self._plotRuler(ax, ruler, blackCoefficient, fcoef)

# přidání legendy
ax.grid(lw=0.5*blackCoefficient, linestyle="-",
      color="0.75")
# posunutí mřížky do pozadí (za graf funkce, osy a mřížky
# aby vizuálně nepřekážela (nové)
for line in ax.lines:
   line.set_zorder(3)
# zprůhlednění legendy (nové)
ax.legend(prop={'size': 10*fcoef}, framealpha=0.80)

# export do souboru
canvas = FigureCanvasAgg(fig)
```

```
            canvas.print_png(fileName,
                             facecolor=fig.get_facecolor())

        def _plotRuler(self, axis, ruler, blackCoefficient, fcoef):
            if ruler.type == "x-axis":
                # vodorovná pravítka
                # zobrazení úsečky
                axis.axhline(y=ruler.y, lw=1.5*blackCoefficient,
                             color='k')
                # zobrazení popisku
                if ruler.label:
                    # definice transformace souřadnic
                    # nejdříve je definováno posunutí v soustavě
                    # kde jednotka je palec (posouvá se o 2 typograf. body)
                    offset = transforms.ScaledTranslation(2/72, 2/72,
                                 axis.figure.dpi_scale_trans)
```

```
            # a vytvoří se složená transformace, která určuje
            # že osa x se zadává v relativních souřadnicích vzhledem
            # k levé a pravé ose (od 0.0 do 1.0)
            # a ose y v souřadnicích dat (s posunutím o 2 body nahoru)
            trans = transforms.blended_transform_factory(
                    axis.transAxes, axis.transData + offset)
            axis.text(0.03, ruler.y, ruler.label,
                    transform=trans,
                    va="bottom", fontsize=8*fcoef)
    elif ruler.type == "y-axis":
        # svislá pravítka
        axis.axvline(x=ruler.y, lw=2*blackCoefficient,
                    color='k')
        # TODO: popiska pravítka
    else:
        # izolované body
```

```
        axis.plot([ruler.x], [ruler.y], marker="o",
                color='k',
                ms=3*blackCoefficient)
        # TODO: popisek izolovaného bodu
```

Hlavní komplikací je nutnost explicitní specifikace transformací u popisků. U vodorovných pravítek (ty jsou zatím jediné plně implementovány), musí být osa *y* zadávána v souřadnicích dat (tj. u nás v souřadnicích, jejichž jednotkami je čas provedení), osa *x* musí být definována relativně k levé ose (bez ohledu na souřadnice dat). V našem případě je umístěna ve 3 % šířky grafu. Navíc v ose *y* musí být popisek vysunut mírně nahoru, aby nebyl částečně překryt pravítkem (posunutí se vyjadřuje v absolutních souřadnicích vyjádřených v palcích resp. jejich 1/72 tj. typografických bodech, neboť se týká textu).

Program pro vytvoření grafu je po všech těch přípravách jednoduchý:

```
import numpy as np
from graphPlot import Plot, FGraph, Ruler, sci_notation
from scipy.optimize import curve_fit

def f(x, a,b):
   return a*b**x

points = np.loadtxt("/tmp/sd-benchmark.txt")

popt, pcov = curve_fit(f, points[:,0], points[:,1])

p = Plot("Benchmark", "modules", "seconds",
       defaultDomain=(3, 45), logScale=True)
p << FGraph(lambda x: f(x, *popt), "${0}\\cdot{1:.2f}^x$"
        .format(sci_notation(popt[0], decimal_digits=2),
```

```
                popt[1]))
p << Ruler(y=60, label="minute")
day = 86400
p << Ruler(y=day, label="day")
p << Ruler(y=365.35*day, label="year")
p << Ruler(y=1000*365.35*day, label="millenium")

for x,y in points:
   p << Ruler(x,y)

p.toFile("/tmp/benchmark.png", blackCoefficient=4.0, dpi=200)
```

Načtení dat z textového souboru provádí funkce *loadtext* z modulu *numpy*. Protože se využívá jednoduchý formát sloupců oddělených tabulátorem není potřeba zadávat žádné dodatečné parametry. Výsledkem je dvourozměrné pole, kde počty modulů tvoří jeden sloupec a časy druhý.

Stejně jednoduché je i nalezení aproximační funkce, stačí jediné volání metody *scipy.optimize.curve_fit* z balíku *Scipy* (což je nadstavba nad *Numpy*). Prvním parametrem je parametrizovaná funkce použitá pro aproximaci nelineární metodou nejmenších čtverců. Tato funkce je definována na začátku zdrojáku: musí to být funkce jedné proměnné (zde je to *x*) a libovolného počtu parametrů (zde *a*,*b*). Protože předpokládáme exponenciální růst časů, budeme aproximovat funkcí ab^x. Druhým parametrem jsou nezávislé (*x*-ové hodnoty), zde je to první sloupec (tj. všechny řádky zapsané jako „:“ což je zkratka za řez `0:n:1`, kde n je počet řádků, a sloupec s indexem 0). Třetím jsou závislé (*y*-ové hodnoty) tj. druhý sloupec.

Úkol: Většina jednodušších numerických balíků podporuje jen tzv. lineární regresi, neboť použitá aproximační metoda nejmenších čtverců se nejsnadněji implementuje pro lineární kombinace funkcí a parametrů (tato implementace je navíc při běhu také velmi rychlá a numericky stabilní). Funkce

ab^x není vyjádřitelná jako lineární kombinace tohoto tvaru, ale lineární regresi lze i pro ni po malé transformaci hodnot využít. Vyzkoušejte i toto řešení za použitá funkce *numpy.linalg.lstsq*.

Aproximační (slangově *fitovací*) funkce vrací dvojici polí. První je pole optimálních hodnot parametrů (tj. zde pole [a,b]), druhý je matice odhadu kovariance (lze využít pro odhad chyb).

Funkce je pak vykreslena spolu s několika pomocnými horizontálními pravítky, které vyznačují jisté orientační časové údaje (je málo lidí vnímá svět ve megasekundách) a datovými body (aby bylo lze vidět jak aproximační funkce odpovídá).

Pro formátování zápisu aproximační funkce je použit LaTeX. Ten si bohužel neporadí s výstupním semilogaritmickým formátem čísel (tvaru např. 1.14e-4) a vypisuje jej bez úpravy. Proto jsem použil funkci, která tento zápis převede do formátu TeXové matematiky (tj. do `1.14\cdot 10^{-4}`). Funk-

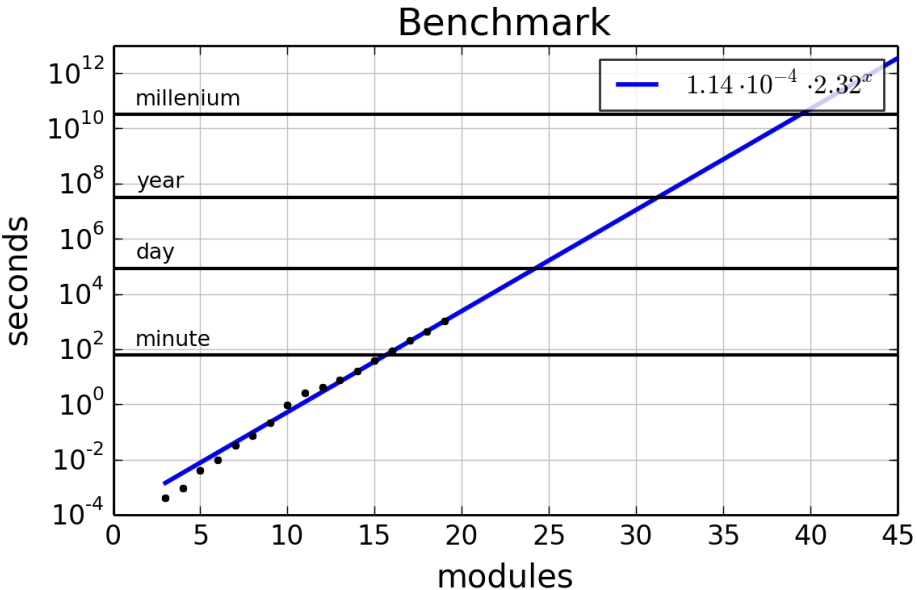
ce je upravena z kódu uvedeného na WWW stránkách

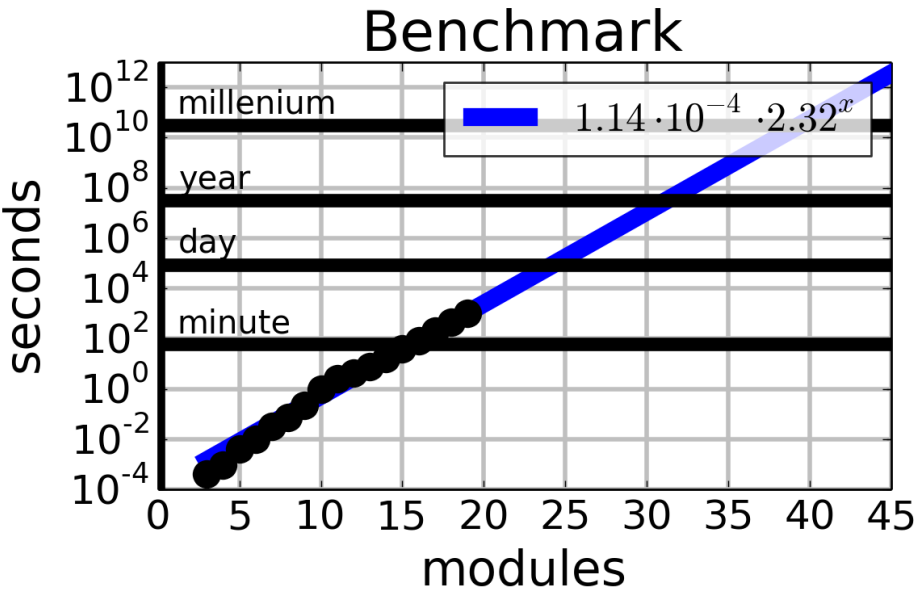
http://stackoverflow.com/questions/18311909/how-do-i-annotate-with-powe
of-ten-formatting

```
def sci_notation(num, decimal_digits=1, precision=None,
                 exponent=None):
    '''
    Returns a string representation of the scientific
    notation of the given number formatted for use with
    LaTeX or Mathtext, with specified number of significant
    decimal digits and precision (number of decimal digits
    to show). The exponent to be used can also be specified
    explicitly.
    '''
    if not exponent:
        exponent = int(floor(log10(abs(num))))
```

```
    coeff = round(num / float(10**exponent), decimal_digits)
    if not precision:
        precision = decimal_digits
    return r"{0:.{2}f}\cdot10^{{{1:d}}}".format(coeff,
                                                exponent,
                                                precision)
```

Výsledek je tento (nejdříve běžná verze a následně verze se zvýrazněnými liniemi):


Benchmark
$1.14 \cdot 10^{-4} \cdot 2.32^{x}$
millenium
year
day
minute
seconds
modules
10^{12}
10^{10}
10^{8}
10^{6}
10^{4}
10^{2}
10^{0}
10^{-2}
10^{-4}
0
5
10
15
20
25
30
35
40
45



Jak je vidět varování před algoritmy před algoritmy s exponenciální časo-

vou složitostí není přehnané. Výpočet aposteriorní pravděpodobnosti pro systém s patnáct moduly trvá řádově minuty (což je ještě akceptovatelné), pro 23 modulů už den (to je ještě realizovatelné), avšak pro třicet modulů přibližně rok (to už je realizovatelné jen stěží) a 39 modulů celé tisíciletí.

Otázka: Vyzkoušejte graf zobrazit v lineární nikoliv logaritmické ose *y*. Jaké jsou výhody a nevýhody tohoto zobrazení.

7. Diagnostika založená na tabulce syndromů

- poznáte jednoduchý algoritmus vhodný pro menší diagnostické grafy
- poznáte význam rozšířené indexace pro snadný zápis maticově orientovaných operací
- implementujete jednoduchý testovací rámce pro testování algoritmů nad složitějšími stavovými prostory

tabulka syndromu, rozšířená indexace, testování algoritmů

7.1 Teorie

Tabulkové algoritmy pracují s **tabulkou syndromu**, což je maticová representace syndromu. Tabulka syndromu $M_R[r_{ij}]$ je čtvercová matice o rozměru $N \times N$, kde N je počet modulů. Pokud je součástí syndromu výsledek

atomické kontroly, kterou provádí i-tý modul na modulu j-tém (to jest hodnota r_{ij}), pak tabulka syndromu obsahuje tento výsledek v i-tém řádku a j-tém sloupci. Položka v tomto případě obsahuje hodnotu nula nebo jedna.

Pokud není atomická kontrola mezi určitými dvěma moduly systému vůbec provedena tj. není v syndromu k dispozici, pak je tato situace graficky representována pomlčku v průsečíku příslušného sloupce a řádku. Při representaci tabulky syndromu v počítači lze použít jakoukoliv hodnotu s výjimkou 0 a 1.

Pokud je graf *t-diagnostikovatelný* (), lze pomocí tabulkových algoritmů identifikovat všechny chybné moduly, ale samozřejmě pouze v případě, že jejich počet nepřekročí t. V opačném případě ($t_A > t$) může být algoritmus schopen tuto situaci detekovat, ale chybné moduly nemohou být identifikovány, tj. program může vypsat pouze oznámení, že počet chybných modulů je příliš velký. Výsledkem však může být i zcela zmatečná identifikace

chybných modulů.

Před volbou a použitím tabulkového algoritmu ja navíc nutné zohlednit resp. zohlednit vlastnosti atomických kontrol. Většina tabulkových algoritmů pracuje s vlastnostmi podle definice 1 na straně 76 (Preparat).

Při volbě konkrétního tabulkového algoritmu je rozhodující počet modulů v systému, neboť tyto algoritmy jsou na počtu modulů silně závislé. Větší počet modulů může algoritmus výrazně zkomplikovat. Navíc může být jejich časová složitost pro větší počet modulů neakceptovatelná (exponenciální) resp. výrazně závislá na charakteru syndromu.

Pro účely výukového textu byl zvolen relativně jednoduchý algoritmus s dobrým chováním pro malé systémy (převzat z [1]). Pro větší systémy ($N > 10$) je také použitelný, ale v určitých situacích může být časově neefektivní (i když pravděpodobnost těchto situací je u reálných systémů velmi malá).

vstupní data algoritmu:

1. **syndrom $R = \{r_{ij}\}$ v podobě tabulky M_R**
2. **hodnota t, typicky je to zároveň optimální hodnota t_{max}**

1.krok

Spočítání celkového počtu jedniček v každém řádku a sloupci. Počet jedniček v řádku s indexem x_i je roven $S_{x_i} = \sum_{j=1}^{n} r_{ij}$, počet jedniček v řádku s indexem y_j je roven $S_{y_j} = \sum_{i=1}^{n} r_{ij}$.

2.krok

Sumy pro řádek a sloupec se stejným indexem se sečtou do jediného součtu tj. $S_i = S_{x_i} + S_{y_i}$ a to pro každé $i = 1 \dots N$. Výsledkem tohoto kroku je tudíž vektor hodnot $S_1, S_2, \dots, S_N$.

3. krok

Každá vypočtená hodnota S_i je porovnána s hodnotou t. Mohou nastat tří situace:

A) $S_i > t$

B) $S_i = t$

C) $S_i < t$

Další krok záleží na tom jaký je výsledek porovnání. Algoritmus se v tomto bodě dělí do tří větví podle hodnoty S_i.

Nejjednodušší je použití větve A (matice je bezprostředně redukována na jednodušší), proto je výhodné nejdříve řešit sloupce a řádky pro něž je $S_i > t$. Teprve ve druhé fázi jsou vyřešeny řádky a sloupce, pro něž $S_i = t$ (podle větve B) a zcela nakonec nejsložitější případ $S_i < t$ (pokud však nějaký takový nevyřešený řádek-sloupec zbude).

Větev A) $S_i > t$

Modul i je určitě chybný.

Důkaz lze provést sporem viz následující obrázek (obrázek je pro konkrétní t a S_i, ale lze jej snadno zobecnit):

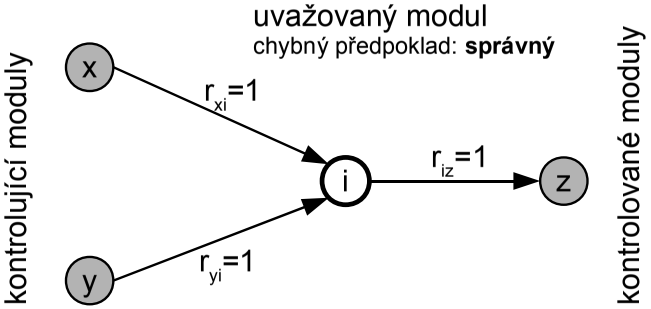
hlavní předpoklady:

$t = 2$

$S_i = 3$ ($S_i > t$)

$M_R =$ (výřez)

	x	i	i	z
x	-		1	
y		-	1	
i	-	-	-	1
z			-	-



musí být chybné
(označují správný modul za chybný)

musí být chybné
(jsou označeny správným modulem za chybné)

spor: tři chybné moduly (x,y,z) $\wedge$ t=2

Modul i se odstraní ze syndromu tj. je nutno vyjmout i-tý řádek a i-tý sloupec. Algoritmus dále pokračuje na redukované tabulce syndromu počínaje prvním krokem.

Větev B) $S_i = t$

Nelze přímo určit **zda je daný modul i chybný nebo nikoliv.**

krok B1: Nejdříve předpokládejme, že je správný (bezchybný)

Tohoto (předběžného) předpokladu můžeme využít pro identifikaci dalších chybných či správných modulů.

krok B2: Jako chybné lze označit moduly, které byly daným (*i*-tým) modulem označeny jako chybné nebo naopak daný modul označily za chybný (i když je dle předpokladu správný). Tj. modul j je chybný, pokud platí: $r_{ij} = 1 \vee r_{ji} = 1$...

krok B3: Podobně lze identifikovat i správné moduly. Správné jsou ty moduly, které byly z *i*-tého modulu kontrolovány s výsledkem 0. Tento postup lze navíc použít i rekurzivně. Na základě předpokladu o správnosti nově identifikovaných modulů, lze předpoklad správnosti rozšířit i na moduly, které byly z těchto modulů kontrolovány s výsledkem 0, a tak rekurzivně dále.

Navíc lze při rekurzivním pohledávání detekovat i chybné moduly (jsou to moduly, které chybně identifikují správné moduly nebo jsou jako chybné těmito správnými moduly označeny). Chybné moduly jsou však listy rekurzivního stromu, nelze je využít pro další rozšiřování prozkoumané sítě).

krok B4: Další krok záleží na tom, zda došlo při rekurzivním procházení podle bodu B2 k rozporu či nikoliv. Rozpor vznikne, pokud dva (předpokládaně) správné moduly otestují jiný modul nekonzistentně (tj. jeden jako správný, druhý jako chybný) resp. pokud (předpokládaně) správný modul

označí moduly získané v kroku B2 jako správné (původní předpoklad je, že jsou chybné).

Pokud není rozpor nalezen, pak jsou všechny předpoklady z kroků B1, B2 a B3 potvrzeny. Pokud ještě zůstane nějaký modul bez potvrzeného odhadu, je možno pokračovat krokem 1 (po eliminaci všech modulů, jejichž předpoklad o správnosti byl potvrzen).

Je-li rozpor nalezen, pak je původní předpoklad B1 nesprávný. Daný modul je označen za chybný a je eliminován z tabulky (jako v kroku A1). Algoritmus pokračuje krokem 1 na redukovaném grafu.

Větev C) $S_i < t$

Pokud nastane tato situace, záleží na celkovém počtu modulů v systému (=N).

Je-li $N < 13$, stačí nalézt sloupec obsahující samé nuly. Tento sloupec od-

povídá modulu, jež lze považovat za správný. S touto informací lze snadno rozhodnout stavy všech modulů (viz krok B3).

Je-li $N \geq 13$, je možno dokázat, že v grafu syndromu existuje cyklus délky $t + 1$, jehož všechny hrany jsou ohodnoceny nulou. Tento cyklus lze nalézt přímo v matici M_R. Všechny moduly representované v grafu uzly nalezeného cyklu jsou správné. Stav ostatních modulů lze odvodit stejně jako v kroku B3.

[konec algoritmu]

7.2 Implementace

Protože algoritmus vychází z tabulky syndromu, začneme implementací metody ***Syndrom.table***, která danou tabulku vytvoří. Tabulky bude stejně jako matice sousednosti representována pomocí dvourozměrné matice poskytované knihovnou ***NumPy***. Důvodem volby je zde možnost využití rozšířené indexace (viz dále). Na druhou stranu není možno na místě neexistujících

atomických kontrol použít hodnotu *None* (která se se sémantického hlediska nejvíce hodí). Běžná *NumPy* pole jsou totiž hodnotová tj. ukládají přímo číselnou hodnotu, nikoliv odkaz na pythonský objekt. Tj. pole s celočíselnými položkami mohou obsahovat jen celá čísla. Pro naše účely zvolíme číslo -1.

```
class Syndrome:
  ...
   def table(self):
      size = self.system.size
      matrix = np.zeros((size, size), dtype=np.int8)
      matrix.fill(-1)
      pos = self.system.mpos # pos je funkce na modulem
      for m_i, m_j, result in iter(self):
         matrix[pos(m_i), pos(m_j)] = result
      return matrix
```

Pole musí být plněno ve dvou krocích (nejdříve se vytvoří s nulovými hodnotami) a pak se plní. Na rozdíl od metody pro vytvoření diagnostického grafu, které se tato metoda (nikoliv překvapivě podobá) je vytvořeno základní dvourozměrné pole nikoliv matice. Důvodem je skutečnost, že částečná indexace polí (tj. získávání jednorozměrných řádků či sloupců) není pro naše účely vhodně definována (za touto krátkou věta je asi hodina ladění a hledání v dokumentaci).

Pro zestručnění zápisu je vytvořena pomocná funkce *pos*, která je ekvivalentní volání metody *mpos* nad systémem daného syndromu. Výsledkem výrazu `self.system.mpos` je funkce jedné proměnné (jíž je libovolný modul systému). Pro teoretiky: vytvoření vázané (*bounded*) metody prostřednictvím uzávěru (*closure*) objektu *self*.

I když lze algoritmus nad tabulkou syndromu realizovat mnoha způsoby, my zvolíme řešení, které nikdy nekopíruje matici. Tím dosáhneme maxi-

mální paměťové a zároveň časové efektivity. Zde to sice není důležité (matice mají rozměr maximálně 20x20), ale může se to hodit pro Vaše další projekty.

```
from selfdiagnosis import Syndrome
from itertools import chain
from benchmark import benchmark
import sys


def detectFaultyModules(syndrome):
    '''
        vyhledá chybné moduly pro systém s daným syndromem.
        Správný výsledek je zaručen, je—li maximálně t chybných
        modulů v t—optimálním grafu a t < t_max.
    '''
```

```
    table = syndrome.table()
    ind = list(range(syndrome.system.size))
    faulty = []

    # informace o využití větví algoritmu
    progress = []

    while ind:
        # hlavní číselné charakteristiky
        n = len(ind)
        t = (n - 1) // 2
        # seznam modulů pro něž lze využít větve B,C
        b_branch = []
        c_branch = []
```

Hlavní funkce *detectFaultyModules* v zásadě kopíruje popis algoritmu. Na začátku jsou vytvořeny klíčové objekty. Proměnná *table* ukazuje na tabulku syndromu (ta se v průběhu algoritmu neměnní), a *faulty* odkazuje seznam chybným modulů (ten je na začátku prázdný). Seznam *ind* obsahuje indexy všech modulů, jejichž stav je ještě neznámý (tzv. seznam platných indexů). Tento seznam určuje, jaká část tabulky je používána v algoritmu detekce — jsou to jen řádky a sloupce, jejichž index je uveden v tomto seznamu. Na počátku je využívána celá tabulka a proto seznam obsahuje všechny indexy od 0 do $n - 1$.

Seznam *progress* nehraje v algoritmu žádnou roli. Je využíván jen ke shromažďování informací o průběhu algoritmu (jaké uzly byly detekovány a v jaké větvi). Tuto informaci lze využít pro ladění a hodnocení algoritmu.

Seznam platných indexů má ještě jednu klíčovou roli — řídí hlavní *while* cyklus. Algoritmus je prováděn, dokud je tento seznam neprázdný (v

Pythonu je neprázdný seznam interpretován v boolovském kontextu jako pravdivý, prázdný je nepravdivý).

Pro každou postupně redukovanou verzi tabulky (seznam platných indexů se v každé iteraci zkracuje) jsou vypočteny základní číselné charakteristiky tabulky – počet modulů (*n*) a hodnota t_{max} (*t*). Jsou také připraveny seznamy pro případné seznamy modulů, které by měly být zpracovány větví B resp. C algoritmu (to se děje jen tehdy neexistuje-li modul řešitelný větví A).

Potom se již prochází jednotlivé nerozhodnuté moduly (tj. indexy v seznamu *ind*):

```
    for i in ind:
        # suma přes sloupce a řádky
        s = sum(chain((x for x in table[i, ind] if x != -1),
                      (x for x in table[ind, i] if x != -1)))
```

```
            if s > t: # větev A
                faulty.append(i)
                progress.append((0, [], [i]))
                _mask((i,), ind, table)
                break
            elif s == t: # potenciální větev B
                b_branch.append(i)
            else: # potenciální větev C
                c_branch.append(i)
```

Nejdříve se vypočítá hodnota s_i. Ta se vypočítá jako suma přes dva zřetězené iterátory, z nichž první prochází řádek s indexem *i* a druhý sloupec s tímto indexem. Při tom se však berou potaz jen sloupce resp. řádky v redukované tabulce, neboť na místě sloupcového resp. indexu je použit seznam platných indexů (jsou vybrány jen ty, jejichž index je v seznamu obsažen).

Zároveň jsou při sumaci vyjmuty buňky s hodnotou -1 (symbolizují neexistenci AT). Díky existenci těchto buněk nelze sumu přes spojené iterátory rozepsat pomocí součtu sum přes každý z nich. Jeden z nich totiž může být prázdný (a suma přes prázdný iterátor není definována).

Poté se již algoritmus větví. V větvi A je i-tý modul přidán mezi chybné a je vyjmut se seznamu platných indexů (voláním pomocné funkce *_mask*). Vnitřní cyklus (*for*) je pak předčasně ukončen.

Je-li $s < t$ pak se prozatím nic neprovádí, jen se přidají indexy modulů pro případné zpracování do příslušných seznamů.

Ty se použijí jen v případě, že není nalezen žádný modul pro nějž je $s > t$, tj. nelze využít větve A. V tomto případě skončí cyklus vyčerpáním iterátoru nad seznamem platných indexů. A jen v tomto případě se provede větev *else* cyklu *for* (není to větev příkazu *if*). Jak již bylo výše uvedeno, Python podporuje větve *else* i u cyklů a zde se to právě hodí.

```
        else: # pokud není cyklus ukončen pomocí break
            if b_branch: # větev B
                hok, hfaulty = _detectFromOK(b_branch[0],
                                             ind, table, False)
                _mask(hok, ind, table)
                _mask(hfaulty, ind, table)
                faulty.extend(hfaulty)
                progress.append((1, hok, hfaulty))
            else: # větev C
                for j in c_branch:
                    if (sum(x for x in table[ind, j] if x != -1)
                            == 0):
                        # modul s nulovým sloupcem
                        hok, hfaulty = _detectFromOK(j, ind,
                                                     table, True)
```

```
                    _mask(hok, ind, table)
                    _mask(hfaulty, ind, table)
                    faulty.extend(hfaulty)
                    progress.append((2, hok, hfaulty))
                    break
            else: # větev s cyklem AT
                raise NotImplementedError("Hledání cyklu AT")
    return faulty, progress
```

V obou větvích se využívá pomocná funkce *_detectFromOK*, která provádí rekurzivní prohledávání okolí uzlu s indexem *i*. Vrací dvojici seznamů. V prvním jsou indexy modulů, které byly bezesporně identifikovány jako správné, v druhém pak chybné. Indexy s obou seznamů jsou vyjmuty se seznamu platných indexů. Ty chybné jsou vloženy do cílového seznamu.

U větve C se toto prohledávání děje jen u modulů, u nichž je součet ve sloupci roven 0, pokud žádný takový není mělo by dojít na vyhledání cyklu atomických kontrol s výsledkem 0. To však prozatím není implementováno a proto je vyhozena příslušná výjimka (nestačí jen nic nedělat, vnější cyklus by pak byl nekonečný!).

Navíc je metoda *_detectFromOK* ve větvi C volána s jiným čtvrtým parametrem (*True* na rozdíl od *False* u větve B). Tento pomocný parametr určuje, zda je předpoklad o správnosti modulu předem jistý. U větve C tomu tak není a tak prohledávání může skončit jeho odmítnutím, u větve C je naopak jistý (a jeho odmítnutí by znamenalo, že algoritmus či spíše jeho implementace není správná či úplná).

Nyní již zbývá jen uvést pomocné funkce:

```
# maže indexy již detekovaných prvků
# ze seznamu indexů
```

```
def _mask(mind, indexes, table):
    for i in mind:
        indexes.remove(i)


# hledá stavy sousedních modulů, za
# předpokladu, že modul 'start' je bezchybný
def _detectFromOK(start, indexes, table, certainAssumption):
    hok = {start}
    hfaulty = set()

    done = set()
    ready = [start]
    # prohledávání začíná ze startovního uzlu

    while ready:
```

```
i = ready.pop() # vyjme poslední prvek z připravených
done.add(i) # a označí jej jako hotový

for j in indexes:
    if table[j, i] == 1:
        # modul chybně identifikující modul 'i'
        hfaulty.add(j)
    if table[i, j] == 1:
        # moduly označené jako chybné modulem 'i'
        hfaulty.add(j)
    elif table[i, j] == 0:
        # moduly označené jako správné modulem 'i'
        hok.add(j)
        if j not in ready and j not in done:
            ready.append(j)
```

```
        if hok.intersection(hfaulty):
            # při nekonzistentním hodnocení, měníme původní
            # předpoklad (startovní modul je chybný)
            assert not certainAssumption
            # tato situace by neměla nastat pokud
            # je předpoklad jistý (ve větvi C)
            return set(), {start}
    return hok, hfaulty
```

Zatímco funkce *_mask* je triviální, je nutno metodu *_detectFromOK* trochu popsat.

Jádrem implementace jsou dva seznamy, které určují stav zpracování jednotlivých modulů. V seznamu *ready* jsou všechny indexy modulů, které jsou považovány za správné, avšak nebylo ještě prozkoumány jejich okolí (tj. moduly, které je kontrolují nebo jsou kontrolovány). Na počátku tento se-

znam obsahuje jen index modulu, od kterého prohledávání začíná. V množině *done* jsou naopak uzly, jejichž okolí již bylo prohledáno. Vyhneme se tak použití rekurze, která je zde neefektivní. Místo toho použijeme v zásadě standardní algoritmus prohledávání do šířky nad frontou (frontou je zde seznam *ready*).

Prohledávání pokračuje tak dlouho, dokud zbývají moduly s neprozkoumaným okolím (vnější cyklus *while*). Pro každý modul jsou prohledány jak kontrolující moduly tak kontrolované moduly a jsou vloženy do příslušných množin (*hok*, *hfaulty*, *h* je z hypotetický). Správné moduly jsou navíc vloženy do seznamu modulů připravených ke zpracování, ale samozřejmě jen tehdy pokud již nejsou prozkoumány resp. již vloženy do seznamu připravených (v grafu AT mohou být cykly).

Trochu překvapivé může být označení modulu jako hotového hned na začátku cyklu (bezprostředně po vyjmutí se seznamu připravených). Prohle-

dávání okolí totiž ještě ani nezačalo. Musíme se však uvědomit, že zpracování modulů se neděje paralelně, ale sériově (postupně se zpracovává jeden za druhým v seznamu). Proto nezávisí na jakém místě těla cyklu je index modulu vložen do seznamu již hotových, jedinou podmínkou je, aby se tak stalo před vstupem do další iterace (= zpracování dalšího modulu z fronty). Upřednostnil jsem bezprostřední návaznost na související operaci vyjmutí (přehlednější a méně náchylné k případnému opomenutí), před vnější logikou zpracování (vy však můžete mít jiný názor).

Cyklus může skončit vyčerpáním seznamu připravených modulů. Pokud se tak stane jsou vráceny seznamy nalezených správných i chybných modulů (ty již jsou potvrzené). Na druhou stranu může (alespoň ve větvi B) dojít ke sporu, když je modul detekován zároveň jako chybný i správný. Proto je na konci každé iterace otestována disjunktnost obou množin (množiny podporují i běžné množinové operace jako zde použitý průnik). Dojde-li ke

sporu, je cyklus funkce předčasně ukončena a jako výsledek je vrácen stav jediného modulu (počátečního, který je navzdory předpokladu chybný).

7.3 Použití

Použití detekční funkce si ukážeme na optimálním diagnostickém grafu s t_{max} chybnými moduly ($n = 15, t_{max} = 7$).

```
if __name__ == "__main__":
    n = 15
    from selfdiagnosis import System, PreparatModule
    m = System.generateModules(n, PreparatModule,
                               [1, 3, 5, 7, 8, 11, 12])
    g = System.optimalSampleATList(n)
    s = System(m, g)
    syndrome = s.getSyndrome()
    print(syndrome.table())
```

```
    with benchmark(0, sys.stdout): # malý benchmark
        faulty, p = detectFaultyModules(syndrome)
        # zde je jádro programu
    print("Result")
    print(sorted(faulty)) # indexy pro přehlednost setřídíme
    print(p) # vypíšeme i stručný průběh algoritmu
```

Algoritmus pro tento vstup funguje, neboť vypíše:

```
0 0.007382
Result
[0, 2, 4, 6, 7, 10, 11]
[(0, [], [0]), (0, [], [2]), (0, [], [4]),
(0, [], [6]), (0, [], [7]), (0, [], [10]),
(0, [], [11]),
(2, {1, 3, 5, 8, 9, 12, 13, 14, 15}, set())]
```

Seznam uzlů vypadá trochu jinak, ale to je dáno tím, že ve výstupu jsou uzly indexovány od nuly a v tovární metodě od jedničky. Výpis průběhu ukazuje, že byla 6krát použita větev A a nakonec jednou větev C.

Úkol: Výpis průběhu není příliš přehledný. Implementujte funkci, která jej převede do přehlednějšího tvaru.

Tento jeden výsledek však nic neříká, zda je algoritmus a jeho implementace správná či nikoliv. I když se omezíme na zde použitý optimální diagnostický graf (který samozřejmě není jediný), pak existuje $2^7 = 128$ možných kombinací 0 až $t_{max} = 7$ chybných modulů, pro něž existuje $\sum_{i=0}^{t} t^i$, kde $t \to \frac{n-1}{2}$ možností různých syndromů (pro n liché, $n > 3$).

$$\sum_{i=0}^{t} t^i, \text{kde } t \to \frac{n-1}{2} = \frac{2^{\frac{1-n}{2}-1}(n-1)^{\frac{n-1}{2}+1} - 1}{\frac{n-1}{2} - 1}$$

Pro $n = 15$ existuje 960 800 různých syndromů.

Je proto vhodné otestovat algoritmus pro více vstupů (nejlépe pro všechny, ale to není vždy možné). Proto si vytvoříme softwarový přípravek (nástroj), který nám s testováním pomůže.

7.4 Testování algoritmů

Testování programů tj. algoritmů, datových návrhů a jejich implementací je jednou ze základních činností vývojáře či vývojového týmu. V reálném vývojovém cyklu často zaujímá více časoprostoru než implementace.

Jedním ze zavedených nástrojů ve světě softwarového vývoje je testování jednotek (***unit-testing***). Ten spočívá ve vytváření relativně jednoduchých testovacích metod, pro téměř každou metodu a to u všech aplikačních tříd. Navíc je nutno testovat i některé kombinace metod v rámci celých procesních řetězců.

Tyto testovací metody musí být odděleny od vlastního aplikačního kódu a měly by produkovat přehledné výsledky. Proto je testování jednotek běžně

podporováno specializovanými knihovnami a celými frameworky. Mezi ty jednodušší patří knihovny označované generickým jménem *xUnit*, jež vycházejí z knihovny *SUnit* programovacího jazyka *Smalltalk*. V Pythonu je tento typ knihoven zastoupen standardní knihovnou *unittest* (též PyUnit) a modernější a snadněji použitelnou knihovnou *python.test*. Zajímavý je též standardní modul *doctest*.

Úkol: Zkuste *doctest* využít pro testování jednoduchých pomocných funkcí (jako je *hit, coalesce apod.*)

Pro testování komplexnějších algoritmů, u nichž se předpokládá složitější vstupní stavový prostor (tj. více komplexněji provázaných vstupů s větším počtem možných prvků) běžné nástroje testování jednotek nestačí.

7.5 Procházení stavových prostorů

Proto si naprogramujeme užitečnou funkci a třídu, které usnadní průchod vstupním stavovým prostorem a vytváření přehledných výsledků. Navíc

umožňují snadnou paralelizaci výpočtů na strojích s větším počtem procesorů/jader (testování je tak rychlejší a alespoň částečně je otestována i paralelizovatelnost algoritmů).

```
if __name__ == "__main__":
   from selfdiagnosis import System, PreparatModule
   from syndromTable import detectFaultyModules

   from multiprocessing.pool import Pool

   with Pool() as pool:
      results = [0, 0, 0]
      ta = DGTableTestAssertion()
      for result, msg in pool.imap(
                ta,
                cascadeIterator(
```

```
                    lambda: range(3, 11),
                    lambda n: range(0, (n-1)//2 + 1),
                    lambda n, t: combinations(range(1, n+1), t)
                    )):
            if result > 0:
                print(msg)
            results[result] += 1
    print(results)
```

Ukázáno je již přímo podpora paralelismu na vysoké abstraktní úrovni. Proto je nejdřív vytvořen objekt *Pool*, který sdružuje několik procesů, které se paralelně podílejí na testování. Implicitně je jich stejný počet jako procesorů v systému.

Poté je vytvořena testovací třída, jejíž instance provádí dílčí testy a hodnotí jejich výstup. Tato instance je volána (instance je totiž volatelná) nad

n-ticí vstupních hodnot, jež vznikají vzájemnou kombinací iterátorů. Toto postupné volání transformuje původní iterátor vstupních hodnot na iterátor hodnot výstupních. Tuto transformaci provádí v Pythonu běžně iterátorová komprehenze, nebo alternativně vestavěná funkce *map* (*map* je standardní jméno pro funkci, která aplikuje jinou (unární) funkci na seznam hodnot). Obě tyto alternativy však provádějí volání postupně (sériově) s využitím jediného procesoru. Toto omezení nemá funkce *Pool.imap*, která distribuuje volání mapovací (transformační) funkce na více procesů v daném poolu (ty poté mohou běžet na více procesorech).

Zdrojem iterátoru je funkce, která spojuje n iterátorů do jediného iterátoru, jenž vrací n-tice (každý i-tý prvek n-tice je poskytován i-tým iterátorem). Navíc v okamžiku, kdy je získána hodnota i-tého iterátoru, jsou destruovány a poté vytvořeny nové objekty následujících iterátorů (tj. $(i-1)$-tého až n-tého), přičemž jejich počáteční stav může záviset na aktuálních hodno-

tách prvního až i-tého iterátoru. Lze tak snadno vytvářet kaskády iterátorů, u nichž se změna iterátorů vyšších řádů projeví na změně chodu iterátoru podřízených. Nově vytvořený iterátor může být jen kopií původního (efektem je zdánlivý restart), ale může to být i iterátor zcela jiný.

Protože jsou iterátory vytvářeny až za běhu (a to mnohonásobně a v závislosti na průběžném stavu), nemohou být přímým parametrem sjednocující funkce. Namísto toho jsou parametrem tovární funkce, které příslušné iterátory vytvářejí. Protože každý i-tý iterátor závisí na $i-1$ iterátorech předchozích musí mít tovární funkce příslušný počet parametrů (u prvního 0, u druhého 1, atd.).

V našem případě vnější iterátor iteruje přes hodnoty od 3 do 5 (včetně). Jeho tovární funkcí je bezparametrický lambda výraz (iterátor se vytvoří jen jednou a na ničem nezávisí). Význam iterátoru je zřejmý, poskytuje počet modulů.

Podřízený iterátor iteruje od 0 do $\left\lfloor \frac{n-1}{2} \right\rfloor$. Hodnota předchozího iterátoru je předána v prvním parametru tovární metody (iterátor je vytvořen pro každé nové *n* znovu). Hodnoty tohoto iterátoru označují počet chybným modulů (od 0 do t_{max} včetně)

Nejspodnější iterátor vytváří všechny kombinace t prvků z množiny indexů $1, 2, \ldots, n$, tj. všechny přípustné kombinace chybných modulů. Jeho tovární lambda funkce závisí na obou předchozích iterátorech.

Pro jistotu si uveďme výpis všech n-tic vracených celým kaskádním iterátorem.

```
3,0,()
3,1,(1,)
3,1,(2,)
3,1,(3,)
4,0,()
4,1,(1,)
```

```
4,1,(2,)
4,1,(3,)
4,1,(4,)
5,0,()
5,1,(1,)
5,1,(2,)
5,1,(3,)
5,1,(4,)
5,1,(5,)
5,2,(1, 2)
5,2,(1, 3)
5,2,(1, 4)
5,2,(1, 5)
5,2,(2, 3)
5,2,(2, 4)
5,2,(2, 5)
5,2,(3, 4)
5,2,(3, 5)
```

```
5,2,(4, 5)
```

Nad těmito n-ticemi je volán testovací objekt, který ověří zda pro daný počet modulů n, a danou množinu chybných modulů $\{m_i, \ldots, m_t\}$ vrací algoritmus nad tabulkou náhodného syndromu pro fixně daný optimální diagnostický graf správně detekovaný stav modulů. Tuto informaci vrací jako dvojici *(druh ukončení, zpráva)*.

Druh ukončení je číslo: 0,1,2, kde

0 - testovaný kód vrátil správný výsledek

1 - testovaný kód vrátil chybný výsledek

2 - testovaný kód skončil předčasně vyvoláním výjimky

Úkol: Z důvodu stručnosti nebyl využit výčtový typ. Zkuste jej sami naprogramovat a pak využít v rámci celého *benchmark* modulu.

Pokud skončí test s chybou (1,2) pak je vypsána zpráva o chybě. Navíc jsou načítány počty jednotlivých druhů ukončení, a součty jsou na závěr vypsány. V našem případě je vypsán seznam [25,0,0], vše je tedy zatím OK.

Nyní se podívejme na implementaci generátoru kaskádního iterátoru:

```
def cascadeIterator(*args):
   levels = len(args) # počet úrovní iterátorů
   iters = [None] * levels # seznam iterátorů
   values = [None] * levels # seznam hodnot

   direction = 1
   al = 0

   while al >= 0: # dokud není vxčerpán vnější
      if al == levels: # dosažení dna
         yield values[:] # vrátíme kopii
```

```
            direction = -1 # a změníme směr
        elif direction == 1: # cesta dolů
            iters[al] = iter(args[al](*values[:al]))
            # vytvoříme nový iterátor
            values[al] = next(iters[al])
            # a získáme jeho první položku
        elif direction == -1: # cesta nahoru
            try:
                values[al] = next(iters[al])
                # pokusíme se získat další položku
            except StopIteration: # je—li výjimka
                pass # nic neděláme
            else:
                direction = 1 # jinak změníme směr
        al += direction # posun na další
```

Kód je stručný, ale použitý algoritmus vyžaduje zamyšlení. Při pohybu směrem dolů (*direction*=1) jsou vytvářeny nové iterátory, při pohybu nahoru (*direction*=-1) jsou získávány nové hodnoty z iterátorů. Po dosažení dna je vrácena příslušná n-tice.

Všimněte si použití řezů, které umožňují snadno získávat podseznamy. Speciální tvar řezu je využit i pro získání kopie celé n-tice za příkazem *yield*. Zde musí být vrácena kopie nikoliv odkaz. V opačném případě funguje jen neparalelní verze mapování.

Otázka: Proč sdílení objektů seznamů funguje v paralelní, ale nikoliv neparalelní verzi? (nápověda: zkuste zohlednit různé scénáře paralelního využití seznamu).

Implementace třídy pro testování funkce nad dílčím vstupem je o něco delší, ale výrazně jednodušší.

```
class TestAssertion:
    '''
        bázová třída objektů pro testování algoritmů
        nad vnějšími vstupy
    '''
    def __init__(self, sep, headers):
        '''
        konstruktor přijímá formátovací údaje
        pro hlavičky výsledných zpráv.

        sep: oddělovač hodnot dimenzí
        header: sekvence hlaviček pro jedotlivé dimenze
        '''
        self.sep = sep
        self.headers = headers
```

```
def __call__(self, values):
    '''
    obálka nad voláním testovací metody.
    Zachytává výsledky a formátuje řetezce
    se zprávou.

    Navíc připojuje údaj o době provedení.
    '''
    header = self.formatF(values)
    try:
        start = process_time()
        rv = self.assertion(*values)
    except Exception as e: # předčasné ukončení výjimkou
        duration = process_time() - start
        return (2, "{0}\t(E) in {3:.2g} sec with {1}('{2}')"
```

```
                .format(header, e.__class__.__name__,
                    str(e), duration))
        else:
            duration = process_time() - start
            if rv is None: # úspěšné dokončení
                return (0, "{0}\t(OK) in {1:.2g} sec"
                      .format(header, duration))
            else: # neúspěch (chybný výstup)
                self.fail += 1
                return (1, "{0}\t(FAIL) in {2:.2g} with {1}"
                      .format(header, rv, duration))

    def formatF(self, values):
        '''
            formátování hlaviček dimenzí
        '''
```

```
        extHeaders = chain(self.headers, repeat(""))
        return self.sep.join(h + str(v)
                             for v, h in zip(values, extHeaders))

    def assertion(self, *args):
        '''
            základní definice testovací metody. Přejímá vstupní
            parametry a vrací buď 'None' (úspěch)
            nebo text s zprávou o chybě.
        '''
        return None # vždy úspěch
```

Základem je obalení vlastní testované funkce/metody pomocnou metodou, jež je aktivována voláním objektu (zde v rámci mapování). Ta zachycuje výstupy testované metody a generuje výstupní zprávu. Výstupní zprávu lze v odvozených třídách modifikovat na třech úrovních: pomocí specifikace od-

dělovačů v konstruktoru, předefinováním metody pro tvorbu hlavičky (lokalizuje vstupy) a nakonec předefinováním metody *__call__* (nejpracnější, ale lze tak zcela změnit chování).

Odvozená třída pro testovaní diagnostiky nad tabulkou syndromů využívá jen první možnost prostřednictvím volání konstruktoru bázové třídy ve svém konstruktoru. Jinak se soustřeďuje pouze na definici testované metody.

```
class DGTableTestAssertion(TestAssertion):
   def __init__(self):
      super().__init__(":␣", ('n=', 't=', 'faulty='))

   def assertion(self, n, t, faulty):
      faulty = set(faulty) # převedení iterátoru na množinu
      m = System.generateModules(n, PreparatModule, faulty)
      g = System.optimalSampleATList(n)
```

```
        s = System(m, g) # vytvoření testovaného systému
        syndrome = s.getSyndrome()
        dFaulty, _ = detectFaultyModules(syndrome)
        # vlastní algoritmus
        dFaulty = {index + 1 for index in dFaulty}
        # úprava výsledku (jiný počátek indexu)
        if faulty != dFaulty: # je-li rozdíl vrať zprávu
           return ("faulty␣{0}␣vers.␣detected␣faulty␣{1}"
                 .format(faulty, dFaulty))
        # jinak ukonči s implicitní návratovou hodnotou None
```

Závěrem jen ukázka výsledku. Pro prozatím testované DG (od n=3 po n=16), nedošlo k žádné chybě, ani volání neimplementované části kódu, takže výstup byl dost monotónní (zde je jen výřez):

```
n=3: t=0: faulty=() (OK) in 0.0016 sec
```

```
n=3: t=1: faulty=(1,) (OK) in 0.0016 sec
n=3: t=1: faulty=(2,) (OK) in 0.0012 sec
n=3: t=1: faulty=(3,) (OK) in 0.0012 sec
n=4: t=0: faulty=() (OK) in 0.0014 sec
n=4: t=1: faulty=(1,) (OK) in 0.002 sec
n=4: t=1: faulty=(2,) (OK) in 0.0012 sec
n=4: t=1: faulty=(3,) (OK) in 0.0014 sec
n=4: t=1: faulty=(4,) (OK) in 0.0015 sec
n=5: t=0: faulty=() (OK) in 0.0021 sec
n=5: t=1: faulty=(1,) (OK) in 0.0017 sec
n=5: t=1: faulty=(2,) (OK) in 0.0019 sec
n=5: t=1: faulty=(3,) (OK) in 0.002 sec
n=5: t=1: faulty=(4,) (OK) in 0.0022 sec
n=5: t=1: faulty=(5,) (OK) in 0.0028 sec
n=5: t=2: faulty=(1, 2) (OK) in 0.0013 sec
```

Bohužel se mi prozatím nepodařilo, zjistit využití zatím neimplementované podvětve větve C. Ta by měla být podle zdroje [1] využita u některých grafů s $n \geqslant 13$. I tento negativní výsledek však ukazuje, že v implementaci může být chyba, která by se mohla projevit u větších grafů resp. k problémům dojde u jiných (typicky suboptimálních DG).

Úkol: Ověřte funkci algoritmu i u suboptimálních grafů resp. grafů $t > t_{max}$. Bude nalezená množina podmnožinou všech chybných modulů?

8. Diagnostika založená na výpočtu aposteriorní pravděpodobnosti

- naučíte se využívat výpočtů aposteriorní pravděpodobnosti pro diagnostiku stavů jednotlivých modulů
- poznáte možnosti symbolického vyhodnocení výrazů

aposteriorní pravděpodobnost, symbolický výpočet

8.1 Teorie

Výpočet aposteriorní diagnostiky pravděpodobnosti určitého globálního stavu systému (a tím i důvěryhodnosti diagnostiky) lze využít i pro účely lokální diagnostiky stavu modulů. Umožňuje totiž vypočítat aposteriorní pravděpodobnosti stavu jednotlivých modulů při daném syndromu.

Algoritmus navrhli H. Fujiwara a K. Kimoskita.

Výpočet je náročný na prostředky (hlavně, jak již víme, na čas) může však být proveden předem. Za běhu systému stačí jen konzultovat tabulku aposteriorních pravděpodobností pro jednotlivé syndromy. Ta má podobu zobrazení syndromu (= bitového vzoru) na seznam aposteriorních pravděpodobností (jednorozměrné pole celých čísel).

Výpočet asposteriorní pravděpodobnosti bezchybného stavu určitého modulu (pravděpodobnost chybného stavu je doplňkem) má tyto kroky:

1. **specifikace hypotéz všech možných stavů systému (2^n hypotéz). Prostor hypotéz je stejný jako u globálního algoritmu.**
2. **výpočet apriorní pravděpodobnosti pro každou hypotézu $P(H_i)$. Výpočet je stejný jako u globálního algoritmu**
3. **výpočet pravděpodobnosti syndromu při stavu modulů předpokládaného danou hypotézou $P(R/H_i)$. I zde se výpočet neliší.**

4. **výpočet aposteriorní pravděpodobnosti správného stavu pro každý modul v systému. Používá se opět bayesovský, ale tentokrát v trochu jiném tvaru:**

$$P_i^* = \frac{\sum_{H_j \in \mathcal{H}_G(i)} P(H_j) P(R/H_j)}{\sum_{H_j \in \mathcal{H}} P(H_j) P(R/H_j)}$$

kde $\mathcal{H}$ je množina všech hypotéz a $\mathcal{H}_G(i)$ je množina všech hypotéz, v nichž je předpokládán bezchybná stav modulu i.

Výpočet aposteriorních pravděpodobností správnosti jednotlivých modulů je však pouze podkladem vlastní diagnostiky. Hlavním cílem je totiž stejně jako u výše uvedených diagnostických metod identifikace správných i chybných modulů.

V některých případech je identifikace správných a chybných modulů zřejmá. Je tomu v případě, když je zjištěná aposteriorní pravděpodobnost vysoká a

především větší než pravděpodobnost apriorní (modul je správný). Podobně, je-li aposteriorní pravděpodobnost nulová či velmi blízká nule (modul je chybný).

Mohou však nastat i hraniční situace (především jsou-li pravděpodobnosti chybných výsledků atomických kontrol vysoké). V tomto případě je nutno stav modulu klasifikovat jako nerozhodnutelný.

Úkol: Zkuste vytvořit graf funkcí $P_i^*(P_{AT})$ pro jednotlivé moduly ve zvoleném diagnostickém grafu (funkce mohou být zobrazeny za použití jediné dvojice os). Diskutujte výsledky.

Zajímavá situace nastane v případě, že obdrženému syndromu R nebude odpovídat žádná hypotéza (a tudíž všechny aposteriorní pravděpodobnosti budou nulové). V tomto případě se evidentně jedná o konfliktní situaci, jejíž příčinou bývá režim selhání, který neodpovídá našemu předpokladu. Typickým příkladem jsou intermitentní selhání, u nichž se chybný stav modulu

projevuje navenek jen střídavě, tj. modul občas selhává (atomické kontroly jej odhalí), v jiných okamžicích však nikoliv (atomické kontroly jej označí za správný).

8.2 Implementace

Implementace může přirozeně vycházet z výpočtu aposteriorní pravděpodobnosti jednotlivých hypotéz, kterou jsme již provedli v kapitole 6.4 na straně 209.

Výpočet by probíhal podle následujícího schématu:

```
for module in modules:
  #všechny hypotézy, v nichž je modul správný
  for hypothesis in H_G(module):
    p[module] += hypothesis.p_aposteori(syndrom)
```

Navíc by bylo možno tento vztah přepsat pomocí seznamové komprehenze,

která by uvnitř používala sumu nad iterátorem přes všechny hypotézy, v nichž je daný modul správný.

Tento zápis by sice byl elegantní, ale výrazně by zpomaloval již tak pomalý výpočet aposteriorních pravděpodobností. Čas výpočtu by byl přibližně dán funkcí $n.e^{n^2}$, což je sice stále „jen“ exponenciální složitost, ale hranici celodenních výpočtů by byla dosažena již někde kolem čtyř resp. pěti modulů.

Při bližším pohledu je naštěstí zřejmé, že výpočet aposteriorních pravděpodobností všech modulů, lze provést v jediném cyklu nad hypotézami, neboť se v něm počítají všechny hodnoty $P(H_j)P(R/H_j)$. Za cenu ztráty elegance a možnosti znovupoužití již hotové funkce získáme mnohem rychlejší implementaci.

```
class Hypothesis
 ...
 @staticmethod
  def apost_probabilityOfModules(syndrome):
     s = syndrome.system
     m = s.moduleList
     size = len(m)
     prs = [0.0] * size
     suma = 0.0
     for h in Hypothesis.h0(size).genAllHypotheses():
        ph = Π(m[i].P_m if h[i] == 0 else 1-m[i].P_m
             for i in range(size))
        prh = Π(m_i.probabilityOfResult(m_j, r,
                       selfState=MState(h[s.mpos(m_i)]),
                       otherState=MState(h[s.mpos(m_j)]))
```

```
                for (m_i, m_j, r) in syndrome)
        suma += ph*prh
        for i in range(size):
            if h[i] == 0:
                prs[i] += ph*prh
    return [pr/suma for pr in prs]
```

Úkol: Vytvoření nové metody, která s původní metodou sdílí podstatné části kódu je problematické (základní pravidlo programátora: každý kód uvést v programu jen jednou). Zkuste obě metody sjednotit. (možné řešení: zavedení třídy pro množinu hypotéz).

8.3 Symbolický výpočet

Metodu pro výpočet aposteriorní pravděpodobnosti lze stejně jako všechny předchozí algoritmy využívat pro numerický výpočet cílových veličin (v tomto případě seznamu číselných hodnot).

Python však prostřednictvím rozšiřujícího knihovny *SymPy* umožňuje provádět i symbolické výpočty tj. výpočty se symbolickými vyjádřeními proměnných a parametrů. I když možnosti balíku *SymPy* nejsou srovnatelné s matematickými prostředími jako *Mathematica* nebo *Maple*, pro naše účely bohatě postačuje.

Jednoduché použití tohoto balíku je triviální.

```
from sympy import symbols, simplify, latex
# definice symbolických proměnných
P_at, P_a, P_b, P_m = symbols("P_at␣P_a␣P_b␣P_m")
m = System.generateModules(2,
      lambda name, initialState: PATModule(name, initialState,
                P_at=P_at, P_a=P_a, P_b=P_b, P_m=P_m),
      [])
s = System(m, "1-2")
```

Jedinou změnou je nutnost inicializovat příslušné proměnné symbolickými hodnotami (tj. objekty, jež zapouzdřují daný symbol). Nejlepším řešením je hromadné přiřazení do všech dotčených proměnných pomocí funkce *sympy.symbols*. Funkce očekává jako parametr řetězec, který definuje posloupnost identifikátorů a vrací n-tici objektů-symbolů. Tj. zde např. proměnná *P_at* bude obsahovat symbol „P_at“ (jméno proměnné se může od jména symbolu obecně lišit, ale ve většině případů je vhodné udržovat shodu).

Proměnné se symbolickým obsahem pak lze využít ve většině kontextů, kde je očekáváno číslo. Zde jsou symbolické hodnoty předány do generátoru modulů. Identifikátory se znovu opakují v jiném významu, ale pokud zápis správně čtěte, žádné problémy nevznikají (např.: „předej jako pojmenovaný parametr *P_at* hodnotu (globální) proměnné *P_at*, jíž je symbol *P_at*“).

Při prvním pokusu o výpis výsledku pomocí běžného volání metody (zajímá nás jen apost. pravděpodobnost prvního modulu):

```
print(Hypothesis.apost_probabilityOfModules(s.getSyndrome()))
```

dojde k následující chybě (výjimce) [zkráceno]:

```
  File "/home/fiser/Dokumenty/LiClipse␣Workspace/Samodiagnostika/src/hypothesis.py",
       line 130, in <lambda>
    PATModule(name, initialState, P_at=P_at, P_a=P_a, P_b=P_b, P_m=P_m),
  File "/home/fiser/Dokumenty/LiClipse␣Workspace/Samodiagnostika/src/selfdiagnosis.py"
       , line 124, in __init__
    assert (initialState == MState.OK and P_m > 0.0 or
  File "/usr/local/lib/python3.4/dist-packages/sympy/core/relational.py", line 111, in
        __nonzero__
    raise TypeError("symbolic␣boolean␣expression␣has␣no␣truth␣value.")
```

Důvod je zřejmý, podmínka aserce *P_m* > 0.0 je pro symbolické hodnoty v zásadě nevyhodnotitelná, tudíž výsledek je nedefinovaný (v tomto případě bohužel/bohudík nepravdivý tj. aserce ukončí program). Proto je nutno aserci umístit do podmínky (provede se jen tehdy, když je pravděpodobnost reálné nebo celé číslo)

```
class PATModule:
 def __init__(...)
  ...
 if isinstance(P_m, numbers.Real):
    assert (initialState == MState.OK and P_m > 0.0 or
          initialState == MState.FAULTY and P_m < 1.0)
```

Po ošetření této závislosti bychom již měli obdržet výsledek:

```
(1.0*P_m**2 + P_m*(−P_at + 1)*(−P_m + 1))/(1.0*P_m**2 + P_m*(−P_a + 1)*(−P_m + 1)
    + P_m*(−P_at + 1)*(−P_m + 1) + (−P_b + 1)*(−P_m + 1)**2)
```

Tento výsledek je relativně čitelný, ale stále lze poznat, že jej nevytvořil člověk. Zajímavý je výskyt koeficientu 1.0, který lze zdánlivě eliminovat. SymPy však rozlišuje mezi přesnou celočíselnou jedničkou a nepřesnou jedničkou typu *float*. Ta representuje interval $(1-\epsilon, 1+\epsilon)$, tj. obecně číslo

různé od 1, které nelze jednoduše eliminovat.

Při bližším pohledu do kódu a hledání neceločíselných konstantních pravděpodobností, není těžké nalézt viníka:

```
class PATModule:
  ...
  def probabilityOfResult(...)
    ...
     if selfState == MState.OK:
         if otherState == MState.OK:
         # pravděpodobnost 1 pro AT mezi správnými moduly
            return resultProbability(0.0, result)
            # má být 0 nikoliv 0.0 !!
```

Po opravě se výsledek o něco zjednoduší:

```
(P_m**2 + P_m*(−P_at + 1)*(−P_m + 1))/(P_m**2 + P_m*(−P_a + 1)*(−P_m + 1) + P_m
    *(−P_at + 1)*(−P_m + 1) + (−P_b + 1)*(−P_m + 1)**2)
```

Výsledek lze samozřejmě převést do LaTeXovské podoby.

```
result = Hypothesis.apost_probabilityOfModules(
         s.getSyndrome())[0]
print(latex(result))
```

Výsledek je je již relativně přehledný:

$$\frac{P_m^2 + P_m\left(-P_{at}+1\right)\left(-P_m+1\right)}{P_m^2 + P_m\left(-P_a+1\right)\left(-P_m+1\right) + P_m\left(-P_{at}+1\right)\left(-P_m+1\right) + \left(-P_b+1\right)\left(-P_m+1\right)^2}$$

Je to stále ještě dost dlouhé a suboptimální (hlavně zápis $(-P+1)$ oproti stručnějšímu $(1-P)$). Bohužel pořadí členů v polynomech je v SymPy

pevně zadrátované (konstantní člen je vždy poslední) a nelze jej pravděpodobně jednoduše změnit. Proto provedeme substituci zápisů $(1-P)$ na Q (Q je pravděpodobnost doplňkové události). Zatím to umím jen specifikací každé substituce zvlášť:

```
result = result.subs(((1-P_m, Symbol('Q_m')),
                      (1-P_at, Symbol('Q_at')),
                      (1-P_a, Symbol('Q_a')),
                      (1-P_b, Symbol('Q_b'))))
```

Výsledek (po konverzi do LaTeXu pomocí funkce *latex)* je výrazně přehlednější:

$$\frac{P_m^2 + P_m Q_{at} Q_m}{P_m^2 + P_m Q_a Q_m + P_m Q_{at} Q_m + Q_b Q_m^2}$$

Použité zdroje

[1] MASHKOV, Viktor, FIŠER Jiří. *Samokontrola a samodiagnostika na systémové úrovni*. Lviv: Ukrainian Academic Press, 2010, 174 s. ISBN 978-966-322-169-4.

[2] PILGRIM, Mark. *Ponořme se do Python(u) 3: Dive into Python 3*. Praha: Cz.Nic, c2010, 430 s. CZ.NIC. ISBN 978-80-904248-2-1.

[3] PYTHON SOFTWARE FOUNDATION. *Python 3.4.1 documentation* [online]. 2014 [cit. 2014-09-20]. Dostupné z: https://docs.python.org/3/

[4] SCIPY developers. *Numpy and Scipy documentation*. [online]. 2014 [cit. 2014-09-20]. Dostupné z: http://docs.scipy.org/doc/.