České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačové grafiky a interakce

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Matěj Klíma**

Studijní program: Softwarové technologie a management
Obor: Web a multimedia

Název tématu: **Automatické generování testovacích scénářů pro procesy webových informačních systémů**

Pokyny pro vypracování:

Navrhněte a implementujte webovou aplikaci, která bude z modelu procesů webového informačního systému generovat odpovídající testovací scénáře. Systém bude přijímat abstrakci procesů v definovaném vstupním formátu a jednoduchým způsobem je vizualizovat. Poté bude umožňovat generování testovacích scénářů na základě několika variant algoritmu specifikovaných vedoucím práce. Systém bude umožňovat zobrazení a export vygenerovaných scénářů v definovaném výstupním formátu. Pomocí sady testů srovnejte kvalitu řešení pro jednotlivé varianty algoritmu, podle kritérií definovaných vedoucím práce.

Seznam odborné literatury:

Kolář, J.: Teoretická informatika, Česká informatická společnost, 2000
Koomen T. et al.: TMap Next, for result-driven testing. UTN Publishers, 2007
Zambon Z.: Beginning JSP, JSF and Tomcat: Java Web Development. Apress 2012

Vedoucí: Ing. Miroslav Bureš, Ph.D.

Platnost zadání: do konce letního semestru 2016/2017

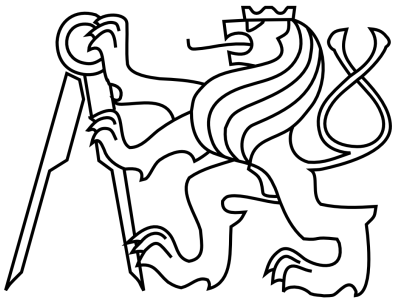
L.S.

prof. Ing. Jiří Žára, CSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 18. 11. 2015

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Bakalářská práce

Automatické generování testovacích scénářů pro procesy webových informačních systémů

Matěj Klíma

Vedoucí práce: Ing. Miroslav Bureš, Ph.D.

Studijní program: Softwarové technologie a management, Bakalářský

Obor: Web a multimédia

25. května 2016

Poděkování

Děkuji vedoucímu mé bakalářské práce panu Ing. Miroslavovi Burešovi, Ph.D. za uvedení do problematiky využití testovacích scénářů a za pomoc s návrhem algoritmů pro jejich generování. Také děkuji studentům bakalářského předmětu TS1, kteří mi pomohli při testování mé aplikace.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 25. května 2016 .....

Abstract

This bachelor thesis focuses on generating test situations, one of software testing techniques. The thesis describes algorithms for automatic generating test situations, both already existing algorithm PCT that implements Process Cycle Test, as well as new algorithms PCT - PA, PCT SmokeTest and PCT SmokeTest II. New algorithms come with less time spent on testing the application and target more important actions thanks to taking the priorities of actions into account when generating the test situations. The thesis also compares effectivity of different algorithms. The algorithms are implemented in web application that allows users to generate test situations in a graph and then download them to their computer.

Abstrakt

Tato bakalářská práce se věnuje generování testovacích scénářů, jedné z technik pro testování softwaru. V práci je obsažen popis algoritmů pro automatické generování testovacích scénářů, a to jak již existujícího algoritmu PCT, který implementuje techniku Process Cycle Test, tak nově implementovaných algoritmů PCT - PA, PCT SmokeTest a PCT SmokeTest II. Nové algoritmy přinášejí, díky zohlednění priorit akcí při generování scénářů, úsporu času nutného pro testování a lepší zacílení na důležité akce. Práce obsahuje také porovnání efektivity jednotlivých algoritmů. Algoritmy jsou implementovány ve webové aplikaci, která umožňuje uživateli v nahraném grafu vygenerovat a stáhnout do počítače testovací scénáře.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Testovací scénáře, kterými se zabývám ve své práci, jsou důležitým prvkem při testování software. Pomocí správně vytvořených testovacích scénářů je možné otestovat každý případ užití v aplikaci, tedy zjednodušeně řečeno je možné otestovat každou interakci uživatele se systémem. Vytvořit testovací scénáře je však u větších informačních systémů komplikované a ne každá firma je na to ochotna vyčlenit čas a finance. Tomu se snažím zabránit svou bakalářskou prací, díky které firmy získají takové scénáře, které nejlépe odpovídají jejich záměrům.

V rámci této bakalářské práce vytvořím webovou aplikaci, která umožní scénáře ze zadaných diagramů automaticky generovat. Na to již sice aplikace existují, ovšem všechny používají pouze algoritmus implementující techniku *Process Cycle Test*. Oproti nim má aplikace umožní scénáře generovat i dalšími algoritmy, které navrhnu s pomocí vedoucího této práce. Motivací pro jejich navržení je vyčlenění akcí, které není potřeba plně testovat, a to díky přiřazení priorit těmto akcím.

U velkých softwarových systémů se může stát, že scénářů, vygenerovaných technikou Process Cycle Test, je tolik, že firma nemá prostředky na to všechny otestovat. Vyřeší to otestováním náhodně vybraných scénářů, což ovšem chyby v implementaci vůbec nemusí odhalit. Oproti tomu stanovení priorit akcím, a použití jednoho z algoritmů implementovaných v mé aplikaci, dá tomuto výběru přesný řád a firmám vědomí, že byl systém správně otestován.

V textu této bakalářské práce uvedu techniky pro ruční vytváření testovacích scénářů, popíši analýzu vytvářené webové aplikace, její implementaci, vysvětlím jak fungují jednotlivé algoritmy a na závěr porovnám efektivitu těchto algoritmů a shrnu jejich přednosti a nedostatky.

Kapitola 2

Základní pojmy

2.1 Graf

Generování testovacích scénářů, čímž se ve své práci zabývám, si lze představit jako vytváření průchodů orientovaným grafem. Proto je nutné zde uvést několik pojmů, které se v orientovaných grafech vyskytují. Definice použitých termínů pak většinou vychází ze skript [1], zavádím zde však i několik vlastních pojmů.

2.1.1 Orientovaný graf, uzel, hrana

„Nechť H a U jsou libovolné disjunktní množiny a σ: H→U×U zobrazení. ***Orientovaným*** [1] ***grafem*** *nazveme uspořádanou trojici G = <H,U,σ>, prvky množiny H nazýváme* ***orientovanými*** [1] ***hranami*** *grafu G, prvky množiny U* ***uzly*** *grafu G a zobrazení σ incidencí grafu G. Incidence přiřazuje každé hraně uspořádanou dvojici uzlů. Jestliže pro h ∈ H je σ(h) = (u, v), nazýváme uzel u počátečním a uzel v koncovým uzlem hrany h. Existenci hrany z u do v vyjadřujeme také zápisem uΓv, uzel u nazýváme* ***předchůdcem uzlu*** *v nebo obdobně uzel v* ***následníkem uzlu*** *u“* [1].

2.1.2 List, kořen, uzel start

„Nemá-li uzel u v orientovaném grafu žádné následníky, nazýváme jej ***listem grafu****. Podobně pro uzel v bez předchůdců používáme označení* ***kořen grafu****“* [1].

Zde je potřeba dodat, že grafy v mé aplikaci nemusí mít žádný kořen, neboť všechny uzly mohou mít nějaké předchůdce. Ovšem pro testovací scénáře je nutné mít v každém grafu definovaný jeden uzel, odkud se vytváří scénář (spojení) do některého z listů. Tomuto uzlu, který bude počátkem všech scénářů, ovšem ne nutně kořenem, budu říkat **uzel start**.

[1] V této práci budu pro jednoduchost u grafů a hran vynechávat slovo „orientovaný“.

2.1.3 Spojení

*„Nechť pro danou dvojici uzlů u a v orientovaného grafu G = <H,U,σ> existuje posloupnost uzlů a hran S = <u_0,h_1,u_1,h_2,...,u_{n-1},h_n,u_n>, kde $h_i \in H$, $\sigma(h_i) = (u_{i-1},u_i)$ pro i = 1,2,...,n, $u_i \in U$ pro i = 0,1,...,n, $u_0 = u$,$u_n = v$. Pak tuto posloupnost nazýváme **spojením grafu** G z uzlu u do uzlu v. Uzel u je počátečním, uzel v koncovým uzlem spojení S, číslo n (≥ 0) nazýváme **délkou spojení**“* [1].

2.1.4 Cyklus

Pro definici cyklu musím nejdříve definovat sled, tah a cestu:

*„Nechť pro danou dvojici uzlů u a v grafu G = <H,U,ρ> existuje posloupnost uzlů a hran S = <u_0,h_1,u_1,h_2,...,u_{n-1},h_n,u_n>, kde $h_i \in H$, $\rho(h_i) = [u_{i-1},u_i]$ pro i = 1,2,...,n, $u_i \in U$ pro i = 0,1,..., n, $u_0 = u$, $u_n = v$. Pak tuto posloupnost nazýváme **sledem grafu** G mezi uzly u a v. Uzly u, v jsou krajní uzly sledu S (u je počáteční, v koncový uzel), uzly u_1,u_2,...,u_{n-1} jsou vnitřní uzly sledu S. Číslo n (≥ 0) nazýváme **délkou** sledu S a značíme d(S). Sled s alespoň jednou hranou, v němž jsou uzly u a v shodné, nazýváme **uzavřeným**, ostatní sledy (včetně sledů nulové délky) nazýváme **otevřenými**“* [1].

*„**Tahem grafu** G nazýváme takový jeho sled, v němž jsou všechny hrany různé. **Cestou grafu** G nazýváme takový jeho tah, v němž každý uzel inciduje nejvýše se dvěma hranami tohoto tahu“* [1].

*„**Orientovaným tahem**, resp. **orientovanou cestou** nazýváme takové spojení, které je tahem, resp. cestou po zrušení orientace. Uzavřenou orientovanou cestu nazýváme **cyklem**“* [1].

2.1.5 Silně souvislá komponenta

*„Orientovaný graf G nazýváme **silně souvislým**, jestliže pro libovolnou dvojici uzlů u,v existuje spojení z uzlu u do uzlu v a spojení z uzlu v do uzlu u (tedy u→v a v→u). **Silnou komponentou** orientovaného grafu nazýváme každý jeho maximální silně souvislý podgraf“* [1].

2.2 Testovací scénáře

Testovací scénář je izolovaný průchod aplikací, ve které testovaný objekt vykazuje specifické chování, které je nutné otestovat. Jinými slovy testovací scénáře jsou jednotlivé možnosti, které je v aplikaci potřeba otestovat [2].

Podmínkou každého scénáře je, že vychází z počátečního bodu, nejčastěji zachycujícího spuštění aplikace, do některého z možných konců průchodu aplikací.

Jednotlivé průchody aplikací tvoří diagram (orientovaný graf), kterému se v softwarovém inženýrství [2] říká UML[3] diagram aktivit[4].

[2]Softwarové inženýrství je činnost věnující se návrhu, tvorbě a údržbě počítačových programů.

[3]Unified Modeling Language (UML) je univerzální grafický jazyk pro vizualizaci návrhu systému

[4]UML diagram aktivit (activity diagram) je pak druh UML diagramu, který se používá pro modelo-

2.2.1 Prioritizace akcí

V průchodech aplikací se mohou vyskytnout akce, jejichž funkčnost je zásadnější, než funkčnost jiných. Ty důležité je pak pochopitelně nutné důkladně otestovat, zatímco ty ostatní se v danou chvíli testovat nemusí. Proto jsem ve své práce implementoval algoritmy, které v průběhu generování testovacích scénářů důležitost těchto akcí zohledňují. Toho jsem docílil ohodnocením hran grafu, který průchody aplikací znázorňuje. Hrany jsou ohodnoceny slovně, prioritami **nízkou** („low“), **střední** („medium“), **vysokou** („high“) nebo **nedefinovanou** („not defined“).

2.3 Techniky pro generování testovacích scénářů

Testovací scénáře se ručně vytváří technikou *Process Cycle Test*. Další technika, kterou lze pro generování použít, byla nazvána *PCT SmokeTest* a od PCT se liší zohledněním priorit jednotlivých akcí. V praxi se však (dle informací od vedoucího práce) navzdory jejím nesporným výhodám zatím používá pouze Process Cycle Test.

2.3.1 Process Cycle Test

Process Cycle Test (PCT) je technika pro vytváření testovacích scénářů, definovaná metodologií TMap (Test management approach) a popsaná v knize [2]. Principem této metody je hledání „testovacích kombinací“ v grafu a jejich spojování do výsledných scénářů.

Princip techniky PCT

Pro každý **větvící bod**, tedy uzel, který má nějaké vstupní a výstupní hrany, v grafu jsou hledány **testovací kombinace** (dále jen kombinace). Ty se skládají z určitého počtu navazujících hran (nebo jedné hrany). Počet navazujících hran v kombinacích je odvislý od **hloubky pokrytí** (někdy také značené zkratkou TDL z anglického test depth level).

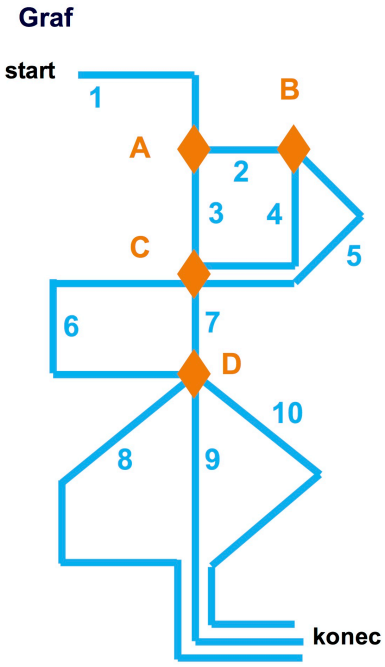
Hloubku pokrytí definoval vedoucí mé práce v díle [4] takto: *Hloubka pokrytí n udává, že pro každý větvící bod se projdou všechny kombinace možných vstupních hran do větvení a n-1 na sebe navazujících výstupních hran.* Hloubku pokrytí volí uživatel a typicky je to hodnota od jedné do tří. Hloubka pokrytí vlastně značí, do jaké hloubky se má sledovat vliv nějaké akce. Z toho plyne, že čím vyšší hodnota bude zvolena, tím větší je pravděpodobnost odhalení chyby na testovaném subjektu. Na druhou stranu s rostoucí hloubkou pokrytí se zvyšuje počet kombinací, z nich testovacích scénářů a kvůli tomu trvá testování déle.

Z vygenerovaných testovacích kombinací se následně sestavují **testovací scénáře**, které musí (v souladu s definicí) začínat v uzlu start a končit v některém z listů grafu. Kombinace se vybírají tak, aby bylo scénářů co nejméně a obsahovaly dohromady co nejméně hran (čím méně je hran ve scénářích, tím se aplikace rychleji otestuje). Toho

vání procedurální logiky, procesů a zachycení workflow aplikace [3].

je dosaženo tím, že se při skládání kombinací do scénářů vyberou vždy ty, které ještě nebyly v žádném scénáři použity (pokud je to možné).

Jak vypadají testovací kombinace a scénáře vygenerované technikou PCT pro určitý graf je zobrazeno na obrázku 2.1. Tento obrázek pochází z díla [4], tedy prezentace předmětu TS1, který vyučuje vedoucí mé bakalářské práce.



Testovací kombinace pro hloubku pokrytí 2

Větvící bod	Kombinace pro test
A	1-2, 1-3
B	2-4, 2-5
C	3-6, 3-7, 4-6, 4-7, 5-6, 5-7
D	6-8, 6-9, 6-10, 7-8, 7-9, 7-10

Testovací scénáře pro hloubku pokrytí 2

Test	Testovací scénáře
1	1-2-4-6-8
2	1-3-6-9
3	1-2-5-6-10
4	1-3-7-8
5	1-2-4-7-9
6	1-2-5-7-10

Obrázek 2.1: *Graf s tabulkami kombinací a scénářů při hloubce pokrytí 2*

2.3.2 PCT SmokeTest

Mnohem menší počet testovacích scénářů, které však stále splňují definici uvedenou v sekci 2.2, se dá vygenerovat technikou nazvanou *PCT SmokeTest*. Tento způsob generování scénářů prozatím není v žádné publikaci přesně definován a jeho princip, který zde popíši, mi vysvětlil vedoucí mé práce. Jeho název se skládá ze dvou částí. Zkratka „PCT“ pochází z pojmu Process Cycle Test, což značí že se touto metodou generují testovací scénáře, „smoke test“ označuje, že se jedná o test nejdůležitějších částí systému[5].

Při generování scénářů technikou PCT SmokeTest je zohledněna priorita jednotlivých akcí, díky čemuž dojde k redukci počtu scénářů. Jaká priorita se má zohlednit je zvoleno

[5]Termín „smoke test“ označuje v oblasti testování softwaru sadu takových testů, které zaručí že nejdůležitější funkce systému fungují správně

kritériem „smoke test level“ (STL). Tedy když je STL nastaven na „high“, generují se scénáře tak, aby v nich byly zahrnuty akce s nejvyšší prioritou. Když je STL nastaven na „medium“, generují se scénáře z akcí s nejvyšší i střední prioritou.

Princip techniky smoke test

Generování scénářů se opět opírá o testovací kombinace, specifikované hloubkou pokrytí (testovací kombinace i hloubku pokrytí vysvětluji v sekci 2.3.1). Oproti PCT však není nutné použít ve scénářích všechny kombinace, ale pouze některé.

Kombinace, které je nutné ve scénářích obsáhnout, budu ve své práci nazývat **prvky množiny PRIO**.

Kombinace patří do množiny PRIO za těchto podmínek:

- Její první hrana má nejvyšší prioritu (v případě, že STL je nastaven na „high“).
- Její první hrana má nejvyšší, nebo střední prioritu (v případě, že STL je nastaven na „medium“).

Pokud je TDL větší než 1, je nutné do množiny PRIO ještě přidat hrany, které:

- Je složena pouze z jedné hrany s nejvyšší prioritou v případě, že STL je nastaven na „high“, a koncovým uzlem této hrany je jeden z listů v grafu.
- Je složena pouze z jedné hrany s nejvyšší prioritou v případě, že STL je nastaven na „high“, nebo střední prioritou v případě, že STL je nastaven na „medium“ a koncovým uzlem této hrany je jeden z listů v grafu.

2.4 Struktura test_scenario

Pro souhrnné označení parametrů, týkajících se právě jednoho běhu některého z implementovaných algoritmů, budu používat termín „TEST_SCENARIO“. Prvek struktury TEST_SCENARIO se bude skládat z následujících údajů:

- Pro jaký graf proběhlo generování scénářů.
- Jaký algoritmus byl použit pro generování.
- Jaká byla hloubka pokrytí.
- Jak se mají interpretovat hrany bez uvedené priority.
- Jaký byl STL (pouze u algoritmů implementujících techniku PCT SmokeTest).
- Jaké byly vygenerovány testovací scénáře.
- Jaké byly vygenerovány testovací kombinace.
- Jak dlouho trvalo generování testovacích kombinací.
- Jak dlouho trvalo generování testovacích scénářů.

Kapitola 3

Analýza

V této kapitole specifikuji jak bude výsledná aplikace realizována a co všechno bude uživateli nabízet.

Nejprve popisuji požadavky [1] na webovou aplikaci, poté model případů užití [2], model klíčového procesu, který při použití aplikace proběhne, a klíčové databázové entity. Na konci kapitoly pak mluvím o uživatelském rozhraní aplikace.

3.1 Požadavky

V této sekci popisuji funkční a kvalitativní [3]požadavky. U každého požadavku uvádím anglickou zkratku typu požadavku, jeho pořadí, název a stručný popis.

3.1.1 Funkční požadavky

FR 1: Import grafu a jeho uložení do databáze

Systém umožní uživateli nahrát na server z jeho počítače graf, který byl vytvořen v aplikaci PCTgen a je ve formátu XML.

FR 2: Zobrazení všech grafů v databázi

Systém umožní uživateli zobrazit seznam všech grafů, které jsou v databázi uloženy.

FR 3: Smazání grafu z databáze

Systém umožní uživateli ze serveru smazat zvolený graf.

[1] Požadavky se v analýze využívají pro lepší odhadnutí časové a finanční náročnosti vývoje systému

[2] Model případů užití umožňuje v systému pojmenovat aktéry, kteří ho budou ovlivňovat, a možnosti, které aktéři budou moci využívat. Každý případ užití pak realizuje určitý / určité funkční požadavky.

[3] Kvalitativní požadavky bývají někdy, ne zcela šťastně, nazývané jako nefunkční.

FR 4: Vizualizace nahraného grafu

Systém umožní uživateli na serveru zobrazit vybraný graf. Vazby mezi uzly a hranami v grafu budou zobrazeny jak textově v tabulce, tak graficky vykreslené v okně prohlížeče.

FR 5: Vytvoření prvku test_scenario

Systém umožní uživateli na serveru vytvořit nový prvek TEST_SCENARIO. Tedy pro zvolený graf vygenerovat testovací scénáře vybraným algoritmem se zadanou hloubkou pokrytí a v případě PCT SmokeTestu i kritériem STL. Webová aplikace také umožní uživateli nastavit, jak se mají při generování scénářů vyhodnocovat hrany bez předem stanovené priority.

FR 6: Smazání prvku test_scenario

Systém umožní uživateli ze serveru smazat prvek TEST_SCENARIO pro daný graf.

FR 7: Vizualizace zvolené kombinace či testovacího scénáře v grafu

Systém umožní uživateli ve vizualizovaném grafu zvýraznit vybranou kombinaci či testovací scénář, tak, aby byla tato sekvence hran v grafu na první pohled patrná.

FR 8: Vizualizace vygenerovaných kombinací a testovacích scénářů v tabulce

Systém umožní uživateli zobrazit všechny testy, které jsou pro daný graf v databázi uloženy. U těchto testů bude zobrazen TDL, případný STL, informace o tom, jak bylo jednáno s hranami bez uvedené priority, všechny vygenerované kombinace a výsledné testovací scénáře.

FR 9: Zobrazení měřících bodů pro porovnání použitých algoritmů

Systém umožní uživateli u každého testu zobrazit také měřící body, které mu umožní porovnat efektivitu použitých algoritmů.

FR 10: Export testovacích scénářů do formátu CSV

Systém umožní uživateli vytvořit a stáhnout dokument ve formátu CSV s vygenerovanými testovacími scénáři pro zvolený test.

3.1.2 Kvalitativní požadavky a omezení

NFR 1: Webová aplikace

Systém bude fungovat jako webová aplikace, přistupovat se k ní tedy bude pomocí webového prohlížeče. Jednotlivé webové stránky budou obsahovat kód v jazyce HTML a skripty v jazyce JavaScript.

NFR 2: Běhové prostředí

Systém bude naprogramován v jazyce Java.

NFR 3: Uživatelské rozhraní

Systém bude ovládán přes grafické uživatelské rozhraní, pomocí myši a klávesnice.

NFR 4: Implementovatelnost algoritmů

Výsledné algoritmy pro generování testovacích scénářů budou snadnou implementovatelné do aplikace PCTgen.

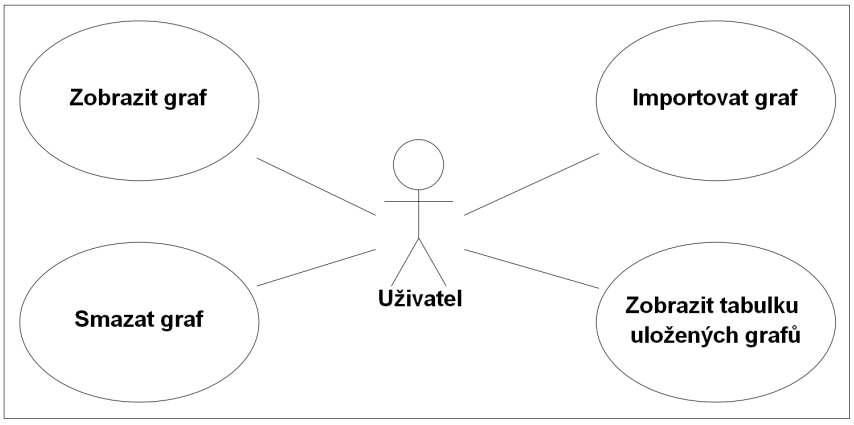
3.2 Případy užití

V této podkapitole popíši nejdůležitější případy užití. Pro přehlednost jsem je rozdělil do tří kategorií. Případy užití v první části se vztahují k entitě „Graf“, v druhé jsou ty, které ovlivňují vizualizaci grafu a ve třetí ty, které se týkají generování testovacích scénářů, tedy entity „Test_scenario“. Všechny tři kategorie obsahují vždy diagram a krátký textový popis jednotlivých případů užití.

Aktéři, kteří v diagramech případů užití figurují, jsou souhrnně označení názvem „Uživatel“. Žádní další nejsou potřeba, neboť každý uživatel aplikace má přístup ke všem jejím funkcím (případům užití).

3.2.1 Práce s grafem

V diagramu 3.1 popisuji případy užití pro manipulaci s grafem.



Obrázek 3.1: *Diagram případů užití zachycující manipulaci s grafem*

Importovat graf

Systém umožní uživateli importovat graf z projektu vygenerovaného aplikací PCTgen a uložit jej do databáze.

Zobrazit tabulku uložených grafů

Systém umožní uživateli v tabulce zobrazit všechny grafy uložené v databázi.

Smazat graf

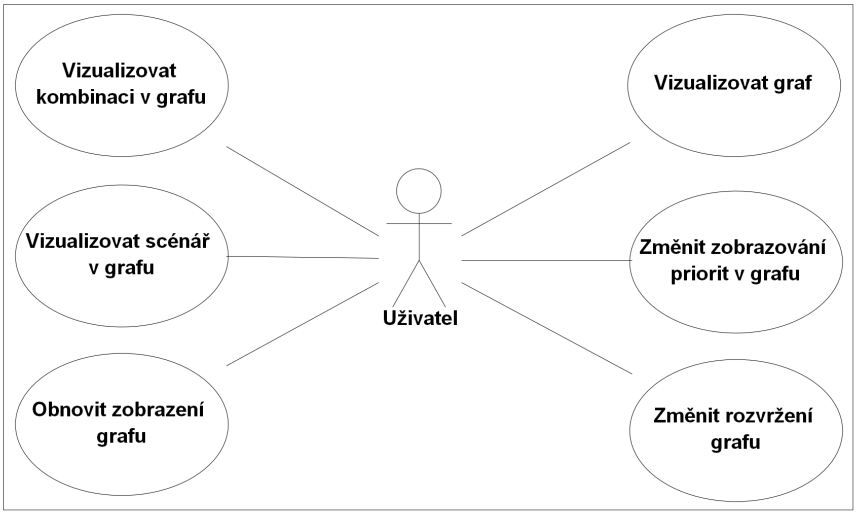
Systém umožní uživateli smazat z tabulky uložených grafů vybraný záznam.

Zobrazit graf

Systém umožní uživateli vybrat graf z tabulky uložených grafů a zobrazit pro něj stránku s generováním testovacích scénářů a vizualizací grafu.

3.2.2 Vizualizace

V diagramu 3.2 popisuji případy užití související s vizualizací grafu.



Obrázek 3.2: *Diagram případů užití zachycující manipulaci s vizualizací grafu*

Vizualizovat graf

Systém umožní uživateli vizualizovat graf a to hned dvěma způsoby.

1. V tabulce, kde pro každý uzel v grafu bude seznam hran, které do něj vstupují a z něj vystupují.

2. Diagramem uzlů a hran s jejich názvy a případně prioritami.

Obnovit zobrazení grafu

Systém uživateli umožní zobrazit graf v původní podobě.

Změnit zobrazování priorit v grafu

Systém umožní uživateli zapnout/vypnout barevné odlišení priorit hran a uzlů.

Změnit rozvržení grafu

Systém umožní uživateli změnit rozvržení hran a uzlů v grafu tak, aby se v něm dokázal co nejlépe orientovat. Také uživateli umožní vybrat si schéma, podle kterého se uzly a hrany budou seskupovat (jako strom, nebo náhodně).

Vizualizovat kombinaci v grafu

Systém umožní uživateli v zobrazeném grafu zvýraznit vybranou kombinaci ze seznamu vygenerovaných kombinací.

Vizualizovat scénář v grafu

Systém umožní uživateli v zobrazeném grafu zvýraznit vybraný testovací scénář ze seznamu vygenerovaných scénářů.

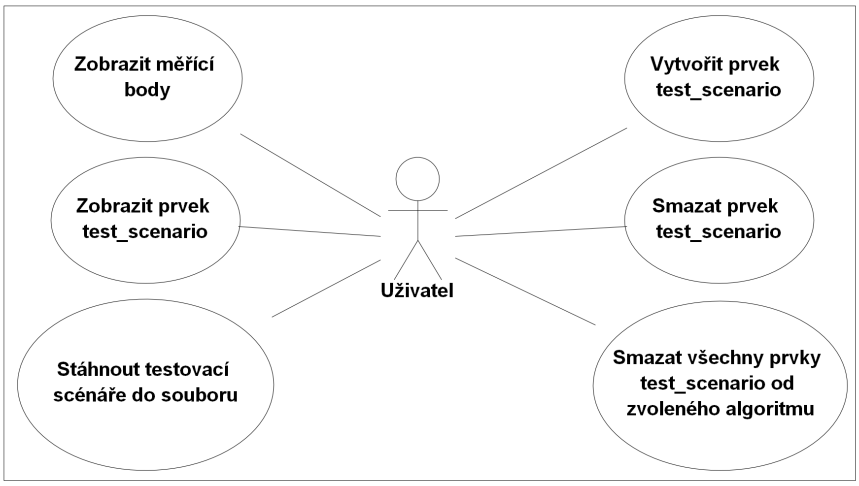
3.2.3 Generování testovacích scénářů

V diagramu 3.3 popisuji případy užití systému související s generováním testovacích scénářů u zvoleného grafu.

Vytvořit prvek test_scenario

Systém umožní uživateli u zvoleného grafu vygenerovat zvoleným algoritmem testovací scénáře. Formulář pro generování testovacích scénářů bude vypadat následujícím způsobem:

1. Výběr algoritmu ze seznamu algoritmů v databázi (formou přepínače)
2. Výběr hloubky pokrytí (čísla)
3. Výběr STL („high“, nebo „medium“)
4. Výběr interpretace hran, u kterých není uvedena priorita (jako hrany s prioritou „low“, „medium“, nebo „high“)



Obrázek 3.3: *Diagram případů užití zachycující generování testovacích scénářů*

Smazat prvek test_scenario

Systém umožní uživateli u zvoleného grafu smazat z databáze vybraný prvek TEST_SCENARIO s vygenerovanými scénáři, kombinacemi apod.

Smazat všechny prvky test_scenario od zvoleného algoritmu

Systém umožní uživateli pro daný graf smazat všechny prvky TEST_SCENARIO vygenerované určitým algoritmem.

Zobrazit prvek test_scenario

Systém umožní uživateli u zvoleného grafu zobrazit z databáze vybraný prvek TEST_SCENARIO s vygenerovanými scénáři, kombinacemi apod.

Stáhnout testovací scénáře do souboru

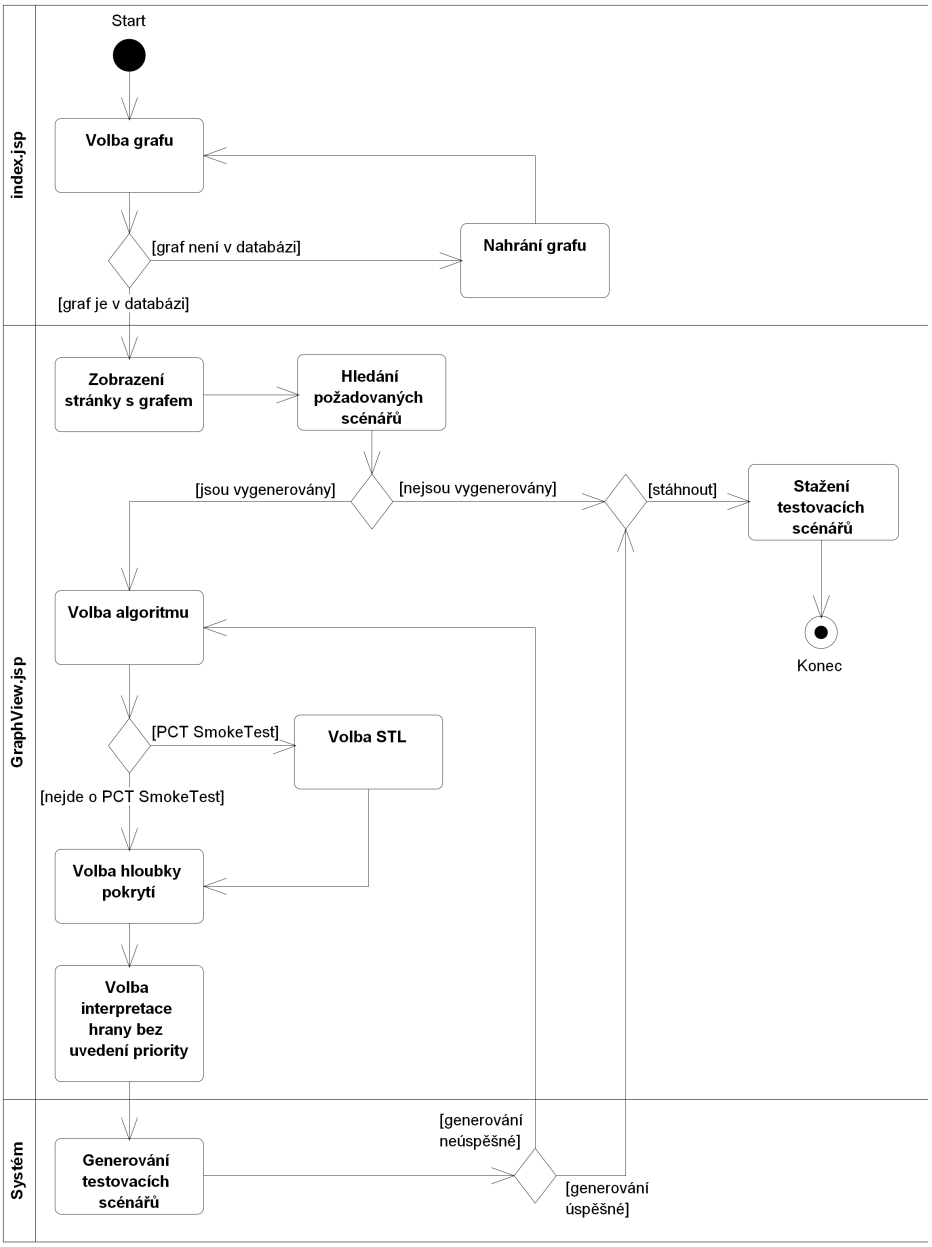
Systém umožní uživateli pro zvolený prvek TEST_SCENARIO stáhnout vygenerované testovací scénáře do souboru ve formátu CSV.

Zobrazit měřící body

Systém umožní uživateli zobrazit pro daný prvek TEST_SCENARIO měření doby generování kombinací, měření doby generování testovacích scénářů, počet kombinací, počet hran v jedné kombinaci, počet hran ve všech kombinacích, počet hran v jednom scénáři, počet hran ve všech scénářích, součet vygenerovaných scénářů a součet hran odpovídajících priorit ve scénářích.

3.3 Základní proces v aplikaci

V této sekci přikládám diagram klíčového procesu v aplikaci, nazvaného „Získání testovacích scénářů“. Diagram, nesoucí označení 3.4, zachycuje průchody aplikací, které může uživatel absolvovat při snaze o získání testovacích scénářů.

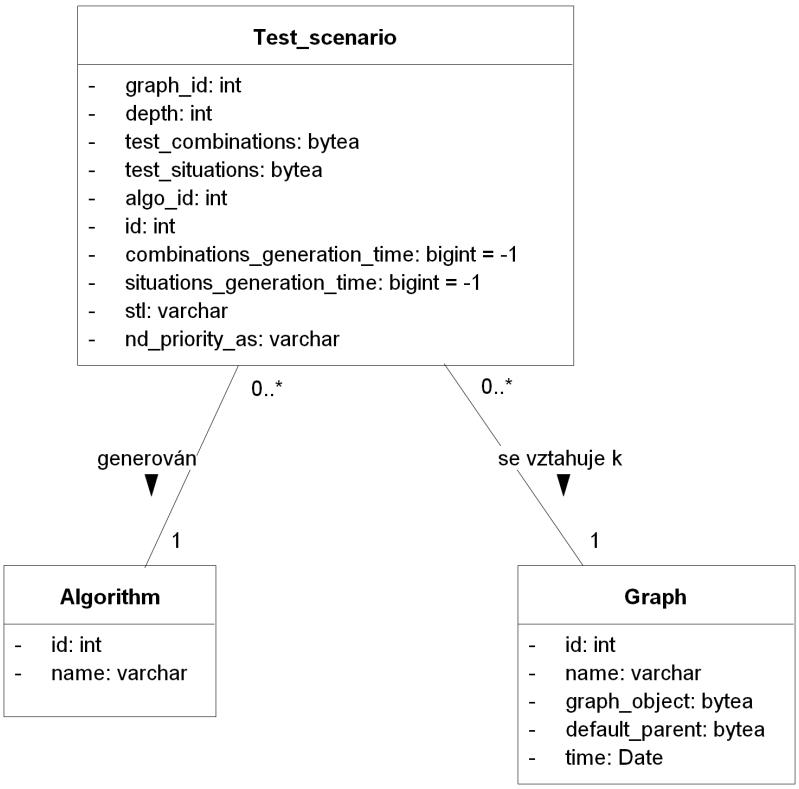


Obrázek 3.4: *Diagram mapující proces získání testovacích scénářů*

3.4 Klíčové entity

Ve své webové aplikaci eviduji tři klíčové entity, které jsou ukládány do odpovídajících tabulek v databázi. Entita `Graph`, pro uložení grafu a jeho klíčových vlastností, entita `TEST_SCENARIO` pro informace o jednotlivých prvcích TEST_SCENARIO (definice tohoto termínu uvedena v úvodu v části 2.4) a entita `Algoritmus`, pro uložení informací o algoritmu, použitelnému pro generování scénářů. Každá entita v datovém modelu obsahuje atribut *id* s jedinečným celočíselným primárním klíčem.

Model vztahů mezi klíčovými entitami zachycuje diagram 3.5.



Obrázek 3.5: *Diagram nejdůležitějších entit a jejich asociací*

3.4.1 Entita Graph

Tato entita eviduje jméno grafu, serializovaný objekt reprezentující graf, objekt s výchozím předkem[4] pro každý uzel a hranu v grafu a čas vložení grafu do databáze.

[4]Díky tomuto objektu typu `mxICell` (viz sekce 4.3.2), který je ve výsledném grafu neviditelný, je možné používat funkce pro vrácení všech uzlů a hran v daném grafu.

3.4.2 Entita Test_scenario

O něco složitější strukturu má entita `TEST_SCENARIO`. Ta obsahuje dva cizí klíče - *graph_id*, odkazujícího se na položku z tabulky entit typu `Graph` a *algo_id* s odkazem na položku z tabulky entit `Algoritmus`. Atribut *depth*, kde je uložena hloubka pokrytí, objekt s výsledkem generování kombinací *test_combinations* a testovacích scénářů *test_situations*, dobu generování kombinací a testovacích scénářů *combinations_generation_time*, respektive *situations_generation_time*, atribut s informací jak se mají interpretovat hrany bez uvedené priority *nd_priority_as* a atribut *stl*, využívaný pouze algoritmy odvozenými od techniky PCT SmokeTest, se zvoleným STL.

3.4.3 Entita Algoritmus

Nejjednodušší entita v modelu `Algoritmus` uchovává kromě *id* už jen informaci o jméně zvoleného algoritmu.

3.5 Uživatelské rozhraní

V této sekci popíšu uživatelské rozhraní hlavní stránky, která bude obsahovat vizualizace zvoleného grafu (tabulkou a diagramem), formuláře a výsledky generování testovacích scénářů. Návrh uživatelského rozhraní, vytvořený pomocí aplikace *Axure RP Pro*, je uveden na obrázku 3.6.

3.5.1 Vizualizace grafu a její parametry

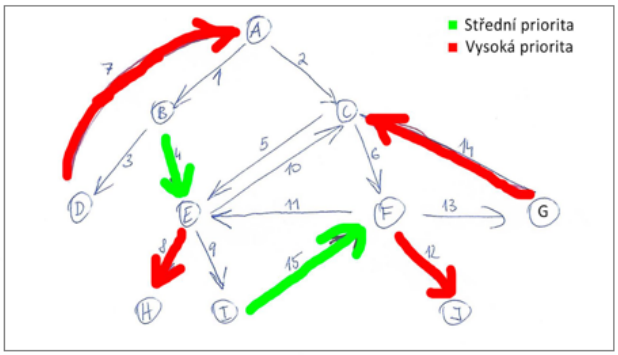
V této části stránky budou vedle sebe oba způsoby pro vizualizaci grafu - diagram i tabulka. Těsně pod touto částí bude formulář, ve kterém se budou dát nastavit parametry tohoto zobrazení, tedy jaké bude rozvržení grafu a zda se mají zobrazovat priority hran.

3.5.2 Formulář pro generování testovacích scénářů

Další částí bude formulář pro generování testovacích scénářů se všemi důležitými parametry, které bude pro daný algoritmus potřeba specifikovat.

3.5.3 Seznam prvků test_scenario

V nejspodnější části bude seznam uložených prvků TEST_SCENARIO. Pro každý algoritmus bude jeden sloupec s odpovídajícími záznamy, které budou skryty v *accordeon* prvku (jeho podoba vysvětlena v sekci 4.1.9). Pro každý algoritmus bude možné najednou zobrazit pouze jeden prvek TEST_SCENARIO, čímž se zlepší přehlednost stránky.



Uzel	Vstup	Výstup
A	7	1, 2
B	1	3, 4
C	2, 10, 14	5, 6
D	...	...

Zobrazit priority

Zobrazení

Grafem Náhodně

Submit

Generovat testovací scénáře

PCT

PCT - PA

PCT SmokeTest

PCT SmokeTest II

Hloubka pokrytí 2

...

Submit

PCT

- TDL 2 Smazat
  - Úvod
  - Kombinace
    - 1 - 4
    - 1 - 5
  - Scénáře Stáhnout
    - 1 - 4 - 10
    - 1 - 5 - 6 - 13
- TDL 3 Smazat

PCT PA

- TDL 1 Smazat
- TDL 2 Smazat
  - Úvod
  - Kombinace
    - 1 - 4
    - 1 - 5
  - Scénáře Stáhnout
    - 1 - 4 - 10
    - 1 - 5 - 6 - 13

PCT SmokeTest

- TDL 1 Smazat
  - Úvod
  - Kombinace
    - 5
    - 15
    - 18
  - Scénáře Stáhnout
    - 1 – 5 – 8 – 9 – 15 – 18
- TDL 3 Smazat

PCT SmokeTest II

- TDL 1
- TDL 3

Obrázek 3.6: *Návrh uživatelského rozhraní*

Kapitola 4

Implementace

V této kapitole je uveden popis technologií, které jsem vybral pro implementaci aplikace. Popisuji zde také architekturu aplikace a některé méně známé datové typy, které využívám.

4.1 Zvolené technologie a knihovny

V této sekci píši o technologiích, které jsem zvolil pro implementaci požadavků z analýzy. Tedy, v jakém programovacím jazyce jsou implementovány jednotlivé algoritmy pro generování testovacích scénářů, jaký webový server jsem použil, jaké jsem vybral technologie pro dynamické generování webových stránek, jaký jsem vybral databázový systém a driver pro práci s ním a jaké jsem využil externí knihovny. Uvádím zde také informace o aplikaci PCTgen a jejím využití v implementaci mé webové aplikace.

4.1.1 PCTgen

PCTgen je desktopová[1] aplikace, která umožňuje uživateli nakreslení grafu v interaktivním editoru a následné vygenerování testovacích scénářů z tohoto grafu dle techniky Process Cycle Test (tato techniky vysvětlena v kapitole 2.3.1) [5]. PCTgen vyvinul pan Lukáš Löwinger v rámci své bakalářské práce na Fakultě elektrotechnické Českého vysokého učení technického v Praze, kterou uvádím v seznamu použité literatury pod číslem [5]. PCTgen je naprogramován v jazyce Java. Grafické uživatelské rozhraní je v něm realizováno pomocí frameworku Swing a s grafy je manipulováno pomocí frameworku JGraphX.

Využití aplikace PCTgen v implementaci

Součástí zadání této bakalářské práce bylo umožnit vytvořené algoritmy snadno včlenit do aplikace PCTgen. Proto pro reprezentaci grafů, hran, uzlů a dalších objektů využívám

[1] Desktopová aplikace je aplikace s grafickým uživatelským rozhraním spuštěná v operačním systému osobního počítače.

stejnojmenné, avšak mírně modifikované[2] třídy z PCTgenu. Tyto třídy jsou uložené v balíčku `pctgen.structures`.

Další podmínkou zadání bylo převzít z PCTgenu algoritmus pro generování testovacích scénářů, nazvaný PCT, který implementuje techniku Process Cycle Test (více o tomto algoritmu v sekci 5.2). Třídy s touto implementací se nachází v balíčcích `algorithms.PCT` a `pctgen.validators`. Z implementace algoritmu PCT vychází, v souladu s instrukcemi od vedoucího této práce, i některé funkce využité algoritmem PCT - PA. Ve všech algoritmech využívám z implementace algoritmu PCT funkci pro generování kombinací createAllCombinationsForNodes a její závislosti. Pro sečtení výskytů sekvencí hran v určitém seznamu využívám z implementace algoritmu PCT funkci countSubSet.

Z aplikace PCTgen jsem převzal také funkce pro načtení grafu ze vstupního souboru do Java objektů a export testovacích scénářů do CSV formátu. Třídy s těmito funkcemi se nachází v balíčku `pctgen.utils`, `pctgen.locale` a `pctgen.exceptions`.

4.1.2 Java

Vzhledem k tomu, že součástí zadání byla podmínka, aby se výsledné algoritmy daly integrovat do desktopové aplikace PCTgen (více o této aplikaci v sekci 4.1.1), napsané v programovacím jazyce Java, zvolil jsem pro implementaci taktéž tento jazyk, konkrétně pak platformu Java EE.

Java je jak programovací jazyk, tak platforma. Jako programovací jazyk je Java vyšší objektově orientovaný jazyk s přesně definovanými pravidly syntaxe a sémantiky. Java platforma je pak specifické prostředí, na kterém aplikace napsané v programovacím jazyce Java přímo běží, nazvané Java Virtual Machine (JVM). Pro spuštění programu v jazyce Java, jehož zdrojový kód byl napsán do souboru s příponou `.java`, v JVM je nutné zdrojový kód zkompilovat do byte kódu, tedy jazyka, kterému JVM rozumí, a uložit ho do stejnojmenného souboru s příponou `.class` [6].

Díky překladu zdrojového kódu do jazyka, kterému rozumí JVM a ne konkrétní počítač, je možné programy napsané v Javě spustit na široké škále přístrojů od domácích počítačů, mobilů, serverů až po SIM karty, stačí k tomu mít na daném stroji spuštěnou JVM. Vzhledem k odlišným výpočetním i paměťovým parametrům těchto přístrojů jsou však v souborech programů, které spouštění programů obstarávají určité rozdíly a nelze tedy spustit stejnou aplikaci na serveru a na mobilním telefonu. Proto se Java ještě rozděluje do následujících kategorií (Java platforem) [7]:

- **Java Standard Edition (Java SE)** pro vývoj aplikací převážně pro osobní počítače.
- **Java Enterprise Edition (Java EE)** pro robustní webové a podnikové aplikace.
- **Java Micro Edition (Java ME)** pro aplikace na vestavěných systémech, mobilních telefonech, tiskárnách a televizích.

[2] Úpravy, které jsem na zmíněných třídách prováděl, spočívaly především v odstranění prvků určených pro framework Swing, který pro účely webové aplikace nelze použít

- **Java Card** pro platební karty, SIM karty a podobně malé přístroje; nejmenší ze všech platforem.

V rámci tohoto dělení je pak vysoce pravděpodobné, že například aplikaci napsanou v Java SE budu moci spustit na jakémkoli osobním počítači bez ohledu na jeho operační systém.

o výhodách v použití jazyku Java k programování aplikací svědčí také fakt, že je v současné době (duben 2016) Java, dle indexu TIOBE, nejpoužívanějším programovacím jazykem vůbec [8].

4.1.3 Servlety a JavaServer Pages

Pro zobrazení grafů, vygenerovaných testovacích scénářů a dalších informací na webovou stránku používám technologii **JavaServer Pages**. JavaServer Pages (JSP) je technologie pro vývoj dynamicky generovaných webových stránek[3] v jazyce HTML, podobná jazyku PHP a technologii ASP od společnosti Microsoft, avšak využívající jazyka Java [9]. Pro správné porozumění fungování JSP stránkám je nutné vysvětlit jejich vztah s technologií zvanou Servlet.

Servlet

Servlet je Javovský objekt, který přijme na serveru žádost (request) od klienta a v závislosti na jeho podobě vyprodukuje pro klienta odpovídající odpověď (response). Přestože servlety dokáží poslat odpověď na jakýkoli typ žádosti, nejčastěji jsou používány v aplikacích běžících na webovém serveru. Samotná webová stránka v jazyce HTML je pak „vytištěna“ nejčastěji v těle metod doGet(HttpServletRequest request, HttpServletResponse response), respektive doPost(HttpServletRequest request, HttpServletResponse response). Generování HTML kódu ve výše zmíněných metodách vypadá velmi podobně jako klasické tištění textu do konzole pomocí metody System.out.println("") z Javy SE. Text je však tvořen validními elementy jazyka hml a místo do konzole se „tiskne“ do instance třídy `HttpServletResponse`, předané v parametru zvolené metody. Vzhledem k tomu, že je však generování HTML kódu v metodách out.print("") velmi nepřehledné a zdlouhavé, vznikla technologie JSP.

JSP stránky pak mají opačný přístup. Místo toho, aby se v čisté Javě „tisklo“ HTML, jako v Servletech, je v JSP stránkách do klasického HTML vměstnaná Java a výsledný kód je pak do servletu automaticky přeložen.

Získání JSP stránky klientem

Průběh získání JSP stránky od klienta ze serveru je následující:

Java Server obdrží od klienta http žádost o JSP stránku. Ta je dohledána a zjistí se, zda byla od posledního převodu na servlet změněna. Pokud ano, stránka je přelo-

[3]Dynamicky generovaná webová stránka se vyznačují tím, že dokáže změnit zobrazovaný obsah bez nutnosti ručního zásahu do jejího zdrojového kódu, jak je tomu potřeba u statických webových stránek

žena na spustitelný servlet, pokud ne, ke konverzi nedojde[4]. Servlet je následně spuštěn a produkuje HTML výstup, předávaný webovému serveru, který ho předá klientovi [10].

Podoba JSP stránek

JSP stránka se skládá ze dvou částí - statického kódu (například v jazyce HTML) a JSP elementů. JSP elementy se pak dělí na skriptovací elementy, direktivní elementy a akční elementy.

Skriptovací elementy jsou bloky kódu psaného v jazyce Java, který je uvozen specifickou sekvencí znaků. V kódu uvozeném skriptovacími elementy můžeme manipulovat s Javovskými objekty a proměnnými, volat metody a zachytávat výjimky. Tyto elementy se dělí do dalších třech kategorií - skriptletů, výrazových elementů a deklaračních elementů. Skriptlety jsou bloky kódu psané jazykem Java, uvozené jsou značkami `<%` a `%>` a teoreticky na obsah stránky nemusí mít žádný vliv. Oproti tomu výrazové elementy obklopené značkami `<%=` a `%>` uvozují výsledek výrazu, který je přímo vložen do kódu stránky. Deklarační elementy, vložené uvnitř značek `<%!=` a `%>` pak obsahují instanční proměnné sdílené napříč všemi požadavky u dané stránky, díky čemuž zabírají méně místa v paměti [10].

Direktivní elementy jsou zprávy adresované JSP kontejneru (např. Tomcatu) o povaze tvořené stránky, nezbytné pro její převod. Příkladem tohoto typu elementů může být direktivní element „page“ s atributem `contentType="text/HTML"`, který říká že stránka je napsána v HTML nebo `import="java.util.Deque"`, který říká že v kódu používáme balíček `java.util.Deque` [10].

Akční elementy pak mohou používat, upravovat nebo vytvářet objekty, které mohou ovlivnit způsob, jakým jsou data předána na výstup. Příkladem takovéhoto elementu je `include`, který umožní přidat do stránky obsah jiné stránky, nebo `taglib`, který umožní rozšířit počet dostupných JSP elementů [10].

4.1.4 Server Apache Tomcat

Každá stránka, přístupná přes internet, potřebuje ke svému provozu webový server. Ten obstarává příjem a zpracování http žádosti a odpovědi, u Java EE aplikací to však nestačí. U nich je ještě potřeba komponenty, která by spustila Java servlety a která dokáže JSP stránku do servletu přeložit. Všechny tyto tři činnosti v sobě spojují Java EE Servery. Ve své aplikaci využívám Java EE Server Tomcat, který je vyvíjen neziskovou organizací Apache Software Foundation [10].

4.1.5 Databázový systém PostgreSQL

Pro uložení dat ve své webové aplikaci využívám relačního databázového systému PostgreSQL.

[4]Díky tomu, že se JSP stránka nemusí překládat na servlet pokaždé ale pouze pokud byla změněna, činí proces generování http odpovědi v porovnání s ostatními skriptovacími jazyky (např. PHP) rychlejší.

Tento otevřený (open-source) software poskytuje plnou podporu pro všechny databázové funkce (cizí klíče, pohledy, trigery...), datové typy a je ho možné spustit na naprosté většině operačních systémů. PostgreSQL navíc poskytuje také programovací rozhraní pro jazyky C/C++, Java, .Net, Petl, Python, Ruby, Tcl, ODBC a vyniká velmi dobrou dokumentací [11]. Pro manipulaci s databází a jejím obsahem (jednotlivými tabulkami) se používá strukturovaný dotazovací jazyk SQL.

4.1.6 JDBC

Aby aplikace, napsaná v jazyku Java, mohla komunikovat s databází, je zapotřebí JDBC ovladače. JDBC, neboli Java Database Conectivity je technologie pro použití programovacího jazyku Java v komunikaci s databázovými systémy, dostupná ve všech Java platformách. Toto API se dělí do dvou hlavních rozhraní. První - JDBC API, je pro tvůrce Java aplikací a to druhé - JDBC driver API je pro tvůrce ovladačů (driverů) pro databázové systémy [12].

Pro PostgreSQL databáze je JDBC driver k dispozici na jejich webových stránkách a pro jeho správnou funkčnost ho stačí stáhnout a připojit k danému projektu do seznamu závislých knihoven.

4.1.7 JGraphX

JGraphX je knihovna pocházející z rodiny mxGraph, určená pro Java Swing framework[5]. Rodina knihoven mxGraph přináší možnost aplikacím interaktivně zobrazovat diagramy a grafy. V současné době poskytuje mxGraph podporu, kromě aplikací napsaných v jazyce Java formou knihovny JGraphX, také pro JavaScript (pro ten ovšem knihovna není poskytována zdarma), .NET a pro Flash aplikace [13].

Knihovna JGraphX, pro Javu reprezentovaná balíčkem `com.mxgraph`, je široce využita v aplikaci PCTgen a její části se dají najít i v mé aplikaci. Samotná reprezentace grafu a jeho prvků, hran a uzlů, jako Java objektů a v podstatě veškerá manipulace s nimi je činěna právě přes funkce poskytované touto knihovnou.

4.1.8 vis.js

JavaScriptová knihovna vis.js umožňuje v prohlížeči vykreslit mnoho druhů dat, například 2D sloupcový diagram, 3D sloupcový diagram, časovou osu a graf uzlů a hran. Ve své práci využívám funkcionalitu knihovny vis.js pro vykreslení grafu uzlů a hran [14].

Pro správné vykreslení grafu je nutné v něm specifikovat množinu uzlů a hran, přičemž obě množiny jsou reprezentovány JavaScriptovým polem. U uzlů je nutné specifikovat jejich ID a název, u hran ještě navíc odkud a kam směřuje. Pomocí dalších parametrů je možné například nastavit tloušťku hrany, barvu hran a uzlů, styl šipky a styl popisek.

[5]Swing, součást platformy Java SE, je framework pro tvorbu a obsluhu aplikací s grafickým uživatelským rozhraním, napsaných v jazyce Java. V současné době je tato knihovna nahrazována modernějším frameworkem JavaFX

Výsledný graf je možné zobrazit s různými druhy rozložení. První možností je náhodné rozložení, které je ovšem možné nastavit tak, že bude graf pokaždé zobrazován stejně, další možností je hierarchické rozložení, které se snaží vykreslit graf jako strom (tedy kořen je nejvíce nahoře a s každým nárůstem hloubky se zvýší i vertikální vzdálenost od kořene, která je však pro každé dva uzly se stejnou hloubkou stejná).

4.1.9 jQuery

JavaScriptová knihovna jQuery usnadňuje rozšíření HTML stránek o dynamické elementy, jako jsou například animace, zpracování událostí a Ajax [6].

Ve své práci používám jQuery pro realizaci grafického elementu zvaného „accordeon". Tento element se při kliknutí na jeho titulek „roztáhne", zobrazí podrobnější informace, a při opětovném kliknutí na titulek se tyto informace opět skryjí.

4.2 Architektura aplikace

Architektura aplikace se dá rozdělit do několika vrstev, které do jisté míry korespondují s architektonickým vzorem Model - View - Controller.

4.2.1 Model

Třídy v modelu, tedy vrstvě, která má na starosti reprezentaci dat aplikace, v sobě skrývají především samotnou implementaci algoritmů pro generování testovacích scénářů. Jsou zde ale také třídy pro export a import grafů, třída pro export uzlů a hran z Java objektů do formátu srozumitelného pro knihovnu `vis.js` a podobně.

4.2.2 View

Vrstva view, která uživateli prezentuje data z modelu, je složena z několika JSP stránek (více o JSP v sekci 4.1.3).

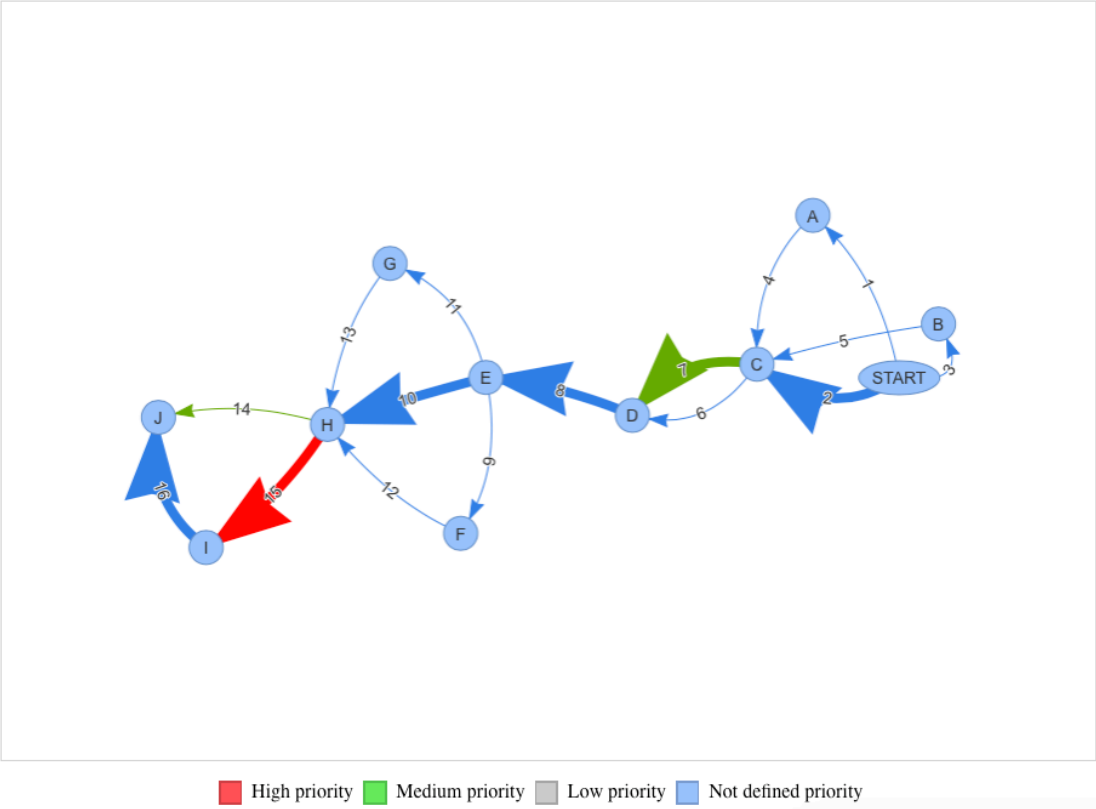
Úvodní stránkou aplikace je *index.jsp* a ta nemá nijak složitou strukturu. Obsahuje jednak formulář s možností nahrát nový graf do aplikace, což obstarává JSP stránka *FileUpload.jsp* a pak také seznam s grafy, které byly již do databáze nahrány. U každého z těchto grafů je pak možnost ho zobrazit, čímž se uživateli zobrazí nejkomplexnější JSP stránka v aplikaci - *GraphView.jsp*.

GraphView.jsp

Tato stránka obsahuje několik částí. Nejprve jsou to prvky vizualizující samotný graf - dynamicky generovaná tabulka a okno s grafem zobrazeným pomocí JavaScriptové knihovny `vis.js`. Pak také několik formulářů, které od uživatele získávají informace, jak se má graf v knihovně `vis.js` zobrazit, jak se mají vygenerovat nové testovací scénáře,

[6] Ajax umožňuje vyslat pomocí JavaScriptu asynchronní http požadavek a výslednou odpověď zobrazit na dané webové stránce bez nutnosti jejího znovu načtení.

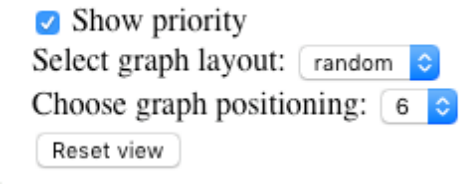
zda se má smazat některý uložený prvek TEST_SCENARIO a zda se mají některé kombinace či situace zvýraznit v grafu (zobrazeno na obrázku 4.1). Nakonec *GraphView.jsp* obsahuje seznam prvků struktury TEST_SCENARIO (definované v sekci 2.4) uložených v databázi, které převádí do přehledných tabulek, skrytých v accordeon prvku.



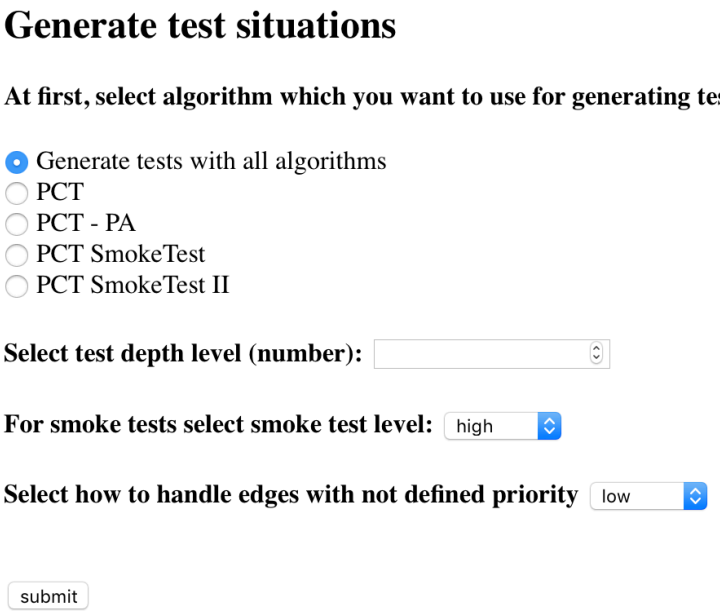
Obrázek 4.1: *Graf se zvýrazněným testovacím scénářem na stránce GraphView.jsp*

4.2.3 Controller

Interakci uživatele s aplikací a instanciaci dat z databáze řídí třídy ve vrstvě controller. Ty jsou v mé aplikaci dvě - *GraphVisualizationController.java* a *TestGenerationController.java*. Obě získávají informace od uživatele z formulářů na stránce *GraphView.jsp*. *GraphVisualizationController.java* o požadavcích na vizualizaci grafu (podoba formuláře je na obrázku 4.2) a *TestGenerationController.java*, o požadavcích na generování testovacích scénářů (podoba tohoto formuláře je na obrázku 4.3) a na mazání TEST_SCENARIO.



Obrázek 4.2: *Formulář s nastavením vizualizace grafu na stránce GraphView.jsp*



Obrázek 4.3: *Formulář s nastavením parametrů pro generování testovacích scénářů na stránce GraphView.jsp*

4.3 Popis vybraných datových typů

4.3.1 Deque

V původní aplikaci PCTgen je pro ukládání seznamů kombinací použito rozhraní (interface) `Deque`, tedy „Double ended queue“. Tato datová struktura umožňuje vkládat i mazat položky na začátek i na konec fronty a dynamicky tak měnit její velikost. Objekty rozhraní `Deque` jsou v této aplikaci představovány datovou strukturou `LinkedList`, tedy spojovým seznamem.

4.3.2 Datové typy z knihovny JGraphX

mxGraph

Objekt tohoto typu reprezentuje graf. Třída `mxGraph` pak poskytuje celou škálu funkcí například pro editaci grafu (například pro přidání a smazání uzlu / hrany) ale také funkce měnící zobrazení grafu frameworkem Swing (například pro změnu pozice uzlu / hrany). Já však grafy needituji, ani nezobrazuji ve Swingu, a proto téměř žádnou funkci nevyužívám.

mxCell

Třída `mxCell` definuje objekt hrany, nebo uzlu v grafu a mnoho funkcí pro práci s nimi. Ve své práci často používám například funkci getTerminal(*source*), která v závislosti na parametru *source* vrátí uzel na počátku, nebo konci hrany. Další užitečnou funkcí je getEdgeCount(), která vrátí počet připojených hran k uzlu. Ve třídě `mxCell` lze nalézt i mnoho funkcí pro editaci grafu, což ovšem ve své práci nepotřebuji.

Ve výsledné aplikaci však většinou používám proměnné typu `mxICell`, což je rozhraní, kterou třída `mxCell` implementuje.

4.3.3 Datové typy z aplikace PCTgen

GraphModel

Mezi třídami z PCTgenu, které ve své aplikaci používám, je nejdůležitější třída jménem `GraphModel`. Ta reprezentuje samotný graf (typu `mxGraph`) a přidává proměnnou *start* (typu `mxICell`) s uzlem start. Třída `GraphModel` navíc poskytuje funkce jako getEdges() a getNodes(), pro získání všech hran, respektive uzlů, v grafu.

GraphEdge a GraphNode

Třídy `GraphEdge` a `GraphNode` definují tzv. „user objekty“, které se vyskytují v buňce grafu (tedy v objektu typu `mxICell`). `GraphEdge` a `GraphNode` obsahují několik metod a atributů, z nichž nejdůležitější pro aplikaci je atribut *priority* typu `String`, který obsahuje prioritu uzlu, nebo hrany. Tyto objekty tedy umožňují uchovat informaci o prioritě uzlů a hran v grafu, což je zcela zásadní pro správnou funkčnost všech nově implementovaných algoritmů.

Kapitola 5

Algoritmy pro automatické generování testovacích scénářů

V této kapitole popisuji čtyři algoritmy pro generování testovacích scénářů, které jsem ve své práci implementoval. Jsou to **PCT**, **PCT - PA**, **PCT SmokeTest** a **PCT SmokeTest II**. Každý zmíněný algoritmus je implementován prostřednictvím několika funkcí, z nichž ty nejdůležitější slovně popisuji a přikládám jejich pseudokód.

Struktura pseodokódů

V pseudokódech dodržuji několik pravidel:

Pseudokód každé funkce je označen popiskem „Algoritmus n“, kde n je pro každý pseudokód unikátní.

Při volání funkcí, jejichž činnost je někde v textu podrobněji popsána, používám bez mezer doslovný název, zakončený kulatými závorkami s parametry (například: CREATETESTSITUATIONS (s, e)). Ostatní funkce jsou volány několikaslovným názvem, skládajícím se z popisu jejich činnosti, který zakončen kulatými závorkami není (například ADD c TO STACK S).

Pseudokódy v některých případech obsahují parametry, které v jejich implementaci chybí. Způsobeno je to tím, že v Javě jsou reprezentovány statickými proměnnými, a tak nemusí být v parametru funkce předávány. Pseudokódy by se tím však stávaly nečitelnější, a proto je mezi vstupními parametry uvádím.

5.1 Struktura implementace algoritmů

Všechny algoritmy, které v této kapitole popisuji, jsou implementovány podobnou strukturou tříd, funkcí, proměnných a jejich názvosloví. Celý proces generování testovacích scénářů vždy řídí funkce GENERATETESTSITUATIONS, jednotlivé testovací scénáře generuje funkce CREATETESTSITUATIONS. K nim jsou pak v pseudokódech pro lepší čitelnost přidány přípony se zkratkou algoritmu, který funkce implementuje (například: GENERATETESTSITUATIONSPCT, GENERATETESTSITUATIONSST apod.).

Počítadlo průchodů

Ve funkcích se často vyskytuje tzv. počítadlo průchodů. Tímto názvem označuji proměnnou, která mapuje hrany (nebo sekvence hran) na celá čísla (v implementaci je to statická proměnná *visitedCounterMap*).

Tuto proměnnou pak využívám pro uchování počtu použití kombinace (nebo hrany) v jednotlivých testovacích scénářích. To je velmi užitečné pro zajištění, že budou při generování scénářů alespoň jednou použity všechny kombinace (což je podmínka techniky PCT).

V algoritmech, které implementují techniku PCT SmokeTest, využívám počítadlo průchodů proti zacyklení při procházení grafem obsahujícím cyklus.

5.2 PCT

Následuje několik slov o algoritmu pro generování testovacích scénářů PCT, který implementuje techniku Process Cycle Test, o které píši v sekci 2.3.1. Tento algoritmus byl obsažen již v aplikaci PCTgen, vytvořené v rámci bakalářské práce [5] (více informací o této aplikaci uvádím v sekci 4.1.1), a já jsem ho v souladu se zadáním mé práce plně převzal.

5.2.1 Popis algoritmu

Kroky tohoto algoritmu kopírují proces popsaný v části 2.3.1. Tedy nejprve jsou podle určité hloubky pokrytí vytvořeny testovací kombinace a z nich se poté skládají testovací scénáře. Scénáře jsou skládány tak, aby se v nich kombinace pokud možno neopakovaly, avšak každá z nich musí být v některém ze scénářů obsažena minimálně jednou.

Rozdělení podle hloubky pokrytí

Generování testovacích scénářů algoritmem PCT je z optimalizačních důvodů rozděleno do dvou částí podle zadané hloubky pokrytí (TDL). První část, reprezentovaná funkcemi, jež mají v názvu uvedeno „ForDepthOne“, je volána při TDL rovnému 1, ostatní jsou volány pro TDL větší než 1. Princip funkcí je naprosto totožný, odlišné je pouze to, že u hloubky pokrytí rovné jedné mají kombinace délku 1 (skládají se pouze z jedné hrany), a tak jsou také reprezentovány datovým typem pro hrany, tedy `mxICell`), oproti tomu při vyšší hloubce pokrytí jsou generovány delší kombinace, které jsou v kódu reprezentovány datovým typem `Deque`. Princip popisovaných funkcí budu vysvětlovat na těch obecných, pro hloubku pokrytí větší než 1. U pseudokódů ale budu dodávat, jaké řádky jsou v alternativních funkcích pro generování scénářů při hloubce pokrytí 1 odlišné.

5.2.2 Řídící funkce

Generování testovacích scénářů řídí funkce GENERATETESTSITUATIONSPCT (Algoritmus 1). Její vstupní parametry jsou m, typu `GraphModel`, a hloubka pokrytí d.

Funkce je rozvětvena do dvou částí - první je použita při hloubce pokrytí rovné jedné, druhá při hloubce pokrytí větší než jedna. V obou částech je však běh podobný - nejdříve je funkcí createAllCombinationsForNodes (Algoritmus 2) vytvořen pro každý uzel seznam kombinací, který je následně pomocí funkce addCombinationToHashTable přeskládán na tabulku kombinací, která je uložena do statické proměnné *combinationsTable*. Kromě seznamu kombinací je funkcí createAllCombinationsForNodes vytvořen také seznam listů, které graf obsahuje a který je uložen do statické proměnné *ends*.

Od každého nalezeného listu je následně volaná funkce createTestSituationsPCT (Algoritmus 4), respektive createTestSituationsForDepthOnePCT při hloubce pokrytí rovné jedné, která z tabulky kombinací vytvoří seznam možných průchodů diagramem od listu do startu (tedy testovací scénáře) tak, aby bylo použito co nejvíc vygenerovaných kombinací, které ještě nebyly použity v ostatních scénářích.

Algoritmus 1 generateTestSituationsPCT(m, d)

1: R := Init variable for resulting list of situations
▷ Vytvoří seznam R, do které se budou ukládat nalezené testovací scénáře
2: V := Init empty map
▷ Inicializuje prázdnou mapu pro počítadlo průchodů, ve které bude klíčem kombinace a hodnotou celé číslo
3: N := All nodes in graph m
▷ Do proměnné N uloží všechny uzly z grafu
4: s := Get start node of graph m
▷ Uloží do proměnné s uzel start
5: **if** $d = 1$ **then**
▷ Pokud je hloubka pokrytí rovna jedné
6: C, E := createAllCombinationsForNodes(d, N)
▷ Zavolá funkci, která vytvoří kombinace, které se uloží do proměnné C, a najde listy, které se uloží do proměnné E
7: **for** $c \in C$ **do**
▷ Pro každou kombinaci
8: T := Add c to table T
▷ Každou kombinaci c zařadí podle koncového uzlu první hrany do tabulky T
9: **end for**
10: **for** $e \in E$ **do**
▷ Pro každý list
11: R' := createTestSituationsForDepthOne(s, e, T, V)
▷ Zavolá funkci pro generování scénářů od uzlu start do procházeného listu
12: R := Add situations R' to list of situations R
▷ Přidá nalezené scénáře do seznamu scénářů
13: **end for**
14: **else**
▷ Pokud je hloubka pokrytí větší než jedna

15: $C, E :=$ createAllCombinationsForNodes(d, N)
▷ Zavolá funkci, která vytvoří kombinace, které se uloží do proměnné C, a najde listy, které se uloží do proměnné E
16: **for** $c \in C$ **do**
▷ Pro každou kombinaci
17: $T :=$ Add c to table T
▷ Každou kombinaci c zařadí podle koncového uzlu první hrany do tabulky T
18: **end for**
19: **for** $e \in E$ **do**
▷ Pro každý list
20: $R' :=$ createTestSituations(s, e, T, V)
▷ Zavolá funkci pro generování scénářů od uzlu start do procházeného listu
21: $R :=$ Add situations R' to list of situations R
▷ Přidá nalezené scénáře do seznamu scénářů
22: **end for**
23: **end if**
24: **return** R
▷ Funkce vrátí výsledný seznam testovacích scénářů

5.2.3 Hledání kombinací

Funkce zmíněné v této sekci používám pro nalezení testovacích kombinací, které jsou definovány v sekci 5.2.3. Testovací kombinace, a tedy i funkce uvedené v této sekci, využívám ve všech implementovaných algoritmech.

Funkce createAllCombinationsForNodes

Funkce createAllCombinationsForNodes (Algoritmus 2) prochází všemi uzly, které jsou jí předány v parametru N; parametr d značí hloubku pokrytí. Pokud procházený uzel nemá potomky, je přidán do seznamu listů E (v implementaci statická proměnná *ends*), a v každém případě je volána funkce goToDepth (Algoritmus 3).

Algoritmus 2 createAllCombinationsForNodes$(d,\ N)$

1: $C :=$ Init variable for resulting list of combinations
▷ Vytvoří seznam C, do kterého se budou ukládat nalezené testovací kombinace
2: $E :=$ Init variable for resulting list of ends
▷ Vytvoří seznam E, do kterého se budou ukládat nalezené listy
3: **for** $n \in N$ **do**
▷ Pro každý uzel ze seznamu uzlů N

```
4:    if The number of children of n = 0 then
   ▷ Pokud uzel nemá potomky
5:        E := Add n to list of ends
   ▷ Přidá uzel do seznamu listů
6:    end if
7:    S := Create new stack
   ▷ Vytvoří nový zásobník pro funkci goToDepth
8:    C := goToDepth(n, d, S, C)
   ▷ Zavolá funkci pro generování kombinací od daného uzlu, výsledné kombinace přidá
   do proměnné C
9: end for
10: return C, E
   ▷ Vrátí výsledný seznam kombinací C a listů E
```

Funkce goToDepth

Funkce goToDepth (Algoritmus 3) rekurzivně prohledává jednotlivé větve od počátečního uzlu až do uvedené hloubky. Přijímá tři parametry, které jsou n, d a S. Proměnná n značí počáteční uzel, d je hloubka pokrytí a S zásobník. Do pseudokódu je navíc přidán ještě parametr C, do které se ukládají nalezené kombinace (v kódu statická proměnná *combinations*). Princip funkce je následující:

(a) Dokud nebyla dosažena požadovaná hloubka pokrytí, prochází se všechny hrany vycházející z uzlu zadaného v parametru. Každá tato hrana je nejdříve přidána na zásobník, následně je na uzel, který z ní vystupuje, zavolána rekurzivně tatáž funkce s hloubkou o jedna menší a stejným zásobníkem, a nakonec se tato hrana ze zásobníku zase odebere.

(b) Pokud byla dosažena požadovaná hloubka pokrytí znamená to, že hrany na zásobníku tvoří dosud nenalezenou kombinaci. Ta je přidána do výsledného seznamu C a funkce vystoupí z rekurze.

Po skončení běhu této funkce jsou v seznamu C hrany v opačném pořadí, než jak jsou ve skutečnosti sestavené v grafu. Výměnu hran do požadovaného pořadí obstarává funkce addCombinationToHashTable, která kromě toho ještě přiřadí každou kombinaci k odpovídajícímu větvícímu bodu (definice větvících bodů uvedena v sekci 2.3.1). Tím vzniká požadovaná tabulka kombinací, která se využívá pro generování kombinací. V implementaci je tato tabulka uložená ve statické proměnné *combinationsTable*, v pseudokódu o ní referuji jako o tabulce T.

Algoritmus 3 goToDepth(n, d, S, C)

1: $d := d - 1$
▷ Sníží hloubku o 1
2: **if** $d < 0$ **then**
▷ Pokud byla dosažena požadovaná hloubka pokrytí
3: $C :=$ From edges in S create new combination and add it to C
▷ Ze všech hran na zásobníku je vytvořena nová kombinace a ta je přidána do seznamu C
4: **return**
▷ Funkce vystoupí z rekurze
5: **end if**
6: $O :=$ Outgoing edges from n
▷ Najde hrany vystupující z n
7: **for** $o \in O$ **do**
▷ Pro každou vystupující hranu o
8: Push edge o to S
▷ Uloží hranu na zásobník
9: $e :=$ Node on the end of edge o
▷ Uloží do proměnné e uzel na konci hrany o
10: goToDepth(e, d, S, C)
▷ Rekurzivní volání funkce
11: Remove last edge from S
▷ Po návratu z rekurze odstraní hranu ze zásobníku
12: **end for**
13: **if** The size of $S = 0$ **then**
▷ Pokud je zásobník prázdný
14: **return** C
▷ Funkce vrátí seznam kombinací C
15: **end if**

5.2.4 Generování scénářů

Pro generování testovacího scénáře je nutné kombinace, získané funkcemi popsanými v předchozí sekci, správně spojit za sebe tak, aby obsahovaly spojení z uzlu start do některého z listů. A aby byly scénáře v souladu se specifikací metody PCT (popsané v části 2.3.1), generují se do té doby, dokud v nich nejsou obsaženy všechny kombinace.

Tuto činnost obstarává funkce createTestSituationsPCT (Algoritmus 4). Jejími vstupními parametry jsou počáteční uzel s a list e. S vygenerovanými kombinacemi funkce v implementaci pracuje přes statickou proměnnou třídy `TMapPCTSituationsGenerator` *combinationsTable*, v pseudokódu přes proměnnou T a statickou proměnnou s mapou použití jednotlivých kombinací ve scénářích *visitedCounterMap* reprezentuje v pseudokódu proměnná V. Po skončení běhu této funkce jsou testovací scénáře uložené ve statické proměnné *testSituations*, v pseudokódu reprezentované proměnnou R.

Princip funkce createTestSituationsPCT

(a) Právě jednou se projdou všechny kombinace.

(b) Pro každou kombinaci se zkusí najít spojení, složené z následujících částí:

  (i) Ze spojení z uzlu start do procházené kombinace (nalezeného funkcí getPathBefore, Algoritmus 5).

  (ii) Ze samotné kombinace.

  (iii) A ze spojení z kombinace do listu (nalezeného funkcí getPathAfter, Algoritmus 6).

(c) Výsledné tři části se uloží do kolekce typu fronta.

(d) Obsah fronty, nalezený testovací scénář, se přidá výsledného seznamu.

Algoritmus 4 createTestSituationsPCT(s, e, T, V)

1: R := Init list for resulting situations
▷ Inicializuje seznam pro testovací scénáře
2: K := Get the keys from T
▷ Vrátí všechny klíče z tabulky kombinací T
3: **for** $k \in K$ **do**
▷ Projde všechny větvící body z tabulky kombinací
4: C := Get the list of combinations for k from T
▷ Vrátí z tabulky T všechny kombinace u klíče k
5: **for** $c \in C$ **do**
▷ Projde všechny kombinace od větvícího bodu
6: v := Get from V how many times was c visited
▷ Zjistí kolikrát byla kombinace použita v některém ze scénářů
7: **if** $v = 0$ **then**
▷ Pokud kombinace ještě nebyla použita v některém ze scénářů
8: **if** Number of edges in $c = 1$ **then**
▷ Pokud je počet hran v kombinaci roven jedné
9: Set c in V as visited
▷ Nastaví kombinaci jako použitou
10: Add c to R
▷ Uloží kombinaci c do seznamu scénářů R
11: **else if** Is it possible to go from e to c **then**
▷ Pokud lze z kombinace dojít do listu
12: Set c in V as visited
▷ Nastaví kombinaci jako použitou
13: P := Init deque
▷ Inicializuje proměnnou pro nové spojení

```
14:                 if Combination c goes not from start s then
    ▷ Pokud kombinace nevede z uzlu start
15:                     b := GetPathBefore(s, c, T, V)
    ▷ Najde spojení z uzlu start do kombinace c
16:                     P := Add b to P
    ▷ Přidá nalezené spojení b do vytvářeného scénáře P
17:                 end if
18:                 P := Add c to P
    ▷ Uloží kombinaci c do vytvářeného scénáře P
19:                 if c is not on the end e then
    ▷ Pokud kombinace nevede do listu e
20:                     a := GetPathAfter(e, c, T, V)
    ▷ Najde spojení z kombinace c do listu e
21:                     P := Add a to P
    ▷ Přidá nalezené spojení a do vytvářeného scénáře P
22:                 end if
23:                 R := Add P do resulting list R
    ▷ Přidá hotový scénář P do seznamu scénářů R
24:             end if
25:         end if
26:     end for
27: end for
28: return R
    ▷ Vrátí seznam s nalezenými scénáři
```

Funkce createTestSituationsForDepthOnePCT

Ve funkci createTestSituationsForDepthOnePCT nejsou příkazy uvedené v pseudokódu Algoritmus 4 na řádcích 8 až 10.

Hledání spojení

Následuje několik slov o funkcích getPathBefore (Algoritmus 5) a getPathAfter (Algoritmus 6). Ty jsou si svou logikou velmi podobné a v obou případech je jejich výsledkem naplněný zásobník. V prvním případě s hledaným spojením z uzlu start v parametru s do kombinace v parametru c, v druhém případě se spojením z kombinace c do listu e. V obou pseudokódech je ještě přidán parametr s tabulkou kombinací T a počítadlo průchodů V.

Princip funkce getPathBefore

(a) Nejprve se z kombinace, do které chceme dojít, odstraní poslední hrana.

(b) Poté se prochází každá kombinace u každého větvícího bodu. Tyto kombinace se

vždy bez první hrany porovnají s kombinací v parametru.

(c) Pokud jsou totožné, procházenou kombinaci lze napojit. Dále se zjistí, zda již byla tato kombinace použita v nějakém scénáři.

(d) Pokud už kombinace někdy použita byla, spustí se metoda GETANOTHERBEFORE, která zkusí najít méně krát použitou kombinaci.

(e) Z nalezené kombinace se uloží na zásobník s hledaným spojením poslední hrana.

(f) Pokud uzel, ze kterého vystupuje poslední hrana na zásobníku, není uzel start, opakuje se průchod cykly a napojování kombinací na hledané spojení uložené v zásobníku.

(g) Funkce vrací zásobník se spojením, které vede z uzlu start do kombinace.

Algoritmus 5 GETPATHBEFORE(s, c, T, V)

```
1: c' := CLONE c WITHOUT LAST EDGE
   ▷ Do nové proměnné okopíruje kombinaci c bez poslední hrany
2: S := INIT STACK FOR RESULTING PATH
   ▷ Inicializuje zásobník pro výsledné spojení
3: K := GET THE KEYS FROM T
   ▷ Vrátí všechny klíče z tabulky kombinací T
4: for k ∈ K do
   ▷ Projde všechny větvící body z tabulky kombinací
5:     L := GET THE LIST OF COMBINATIONS FOR k FROM T
   ▷ Vrátí z tabulky T všechny kombinace u klíče k
6:     for l ∈ L do
   ▷ Pro jednotlivé kombinace
7:         if GET l WITHOUT FIRST EDGE = c' then
   ▷ Pokud se kombinace l bez své první hrany rovná kombinaci c', lze ji napojit před c
8:             v := GET FROM V HOW MANY TIMES WAS l VISITED
   ▷ Kolikrát byla kombinace použita v některém ze scénářů
9:             if v ≠ 0 then
   ▷ Pokud již kombinace l byla použita v některém ze scénářů
10:                l' := GETANOTHERBEFORE(c', v, T, V)
   ▷ Zkusí najít výhodnější kombinaci a přiřadit jí do proměnné l'
11:                if l' ≠ 0 then
   ▷ Pokud byla výhodnější kombinace nalezena
12:                    l := l'
   ▷ Výhodnější kombinace se nastaví jako aktuální procházená
13:                    ADD TO V A VISIT OF l
   ▷ Započítá se použití kombinace l
14:                end if
```

```
15:             else
    ▷ Pokud výhodnější kombinace nebyla nalezena
16:                 Add to V a visit of l
    ▷ Započítá se použití kombinace l
17:             end if
18:             p := Get last edge from l
    ▷ Získá z kombinace l její poslední hranu
19:             S := Push p to S
    ▷ Uloží hranu p do zásobníku pro výsledné spojení S
20:             c' := Get l without last edge
    ▷ Přiřadí do proměnné c' kombinaci l bez její poslední hrany
21:             if Get the source node of first edge in c' = s then
    ▷ Pokud kombinace c' vychází z uzlu start
22:                 return S
    ▷ V zásobníku S se nachází spojení z uzlu start do kombinace c
23:             else
24:                 Repeat from line 6
    ▷ Spojení v zásobníku S nevychází z uzlu start, proces se opakuje
25:             end if
26:         end if
27:     end for
28: end for
```

Princip funkce getPathAfter

(a) Nejprve se ze spojení, do kterého chceme dojít, odstraní první hrana.

(b) Poté se prochází každá kombinace u každého větvícího bodu. Tyto kombinace se vždy bez poslední hrany porovnají s kombinací v parametru.

(c) Pokud jsou totožné, procházenou kombinaci lze napojit. Dále se zjistí, zda už byla tato nalezená kombinace použita v nějakém scénáři.

(d) Pokud už kombinace někdy použita byla, spustí se metoda getAnotherAfter, která zkusí najít méně krát použitou kombinaci.

(e) Z nalezené kombinace se uloží na zásobník s hledaným spojením první hrana.

(f) Pokud uzel, ze kterého vystupuje první hrana na zásobníku, není list v parametru, opakuje se průchod cykly a napojování kombinací na hledané spojení uložené v zásobníku.

(g) Funkce vrací zásobník se spojením, které vede z kombinace do listu.

Algoritmus 6 getPathAfter(e, c, T, V)

```
1: c' := Clone c without first edge
   ▷ Do nové proměnné okopíruje kombinaci c bez první hrany
2: S := Init stack for resulting path
   ▷ Inicializuje zásobník pro výsledné spojení
3: K := Get the keys from T
   ▷ Vrátí všechny klíče z tabulky kombinací T
4: for k ∈ K do
   ▷ Projde všechny větvící body z tabulky kombinací
5:     L := Get the list of combinations for k from T
   ▷ Vrátí z tabulky T všechny kombinace u klíče k
6:     for l ∈ L do
   ▷ Pro jednotlivé kombinace
7:         if Get l without last edge = c' and
   Is it possible to go from end e to combination l then
   ▷ Pokud se kombinace l bez své poslední hrany rovná kombinaci c', a je z ní možné
   dojít do konce, lze ji napojit za c
8:             v := Get from V how many times was l visited
   ▷ Kolikrát byla kombinace použita v některém ze scénářů
9:             if v ≠ 0 then
   ▷ Pokud již kombinace l byla použita v některém ze scénářů
10:                l' := GetAnotherAfter(c', e, v, T, V)
   ▷ Zkusí najít výhodnější kombinaci a přiřadit jí do proměnné l'
11:                if l' ≠ 0 then
   ▷ Pokud byla nějaká výhodnější kombinace nalezena
12:                    l := l'
   ▷ Výhodnější kombinace se nastaví jako aktuální procházená
13:                    Add to V a visit of l
   ▷ Započítá se použití kombinace l
14:                end if
15:            else
   ▷ Pokud výhodnější kombinace nebyla nalezena
16:                Add to V a visit of l
   ▷ Započítá se použití kombinace l
17:            end if
18:            p := Get first edge from l
   ▷ Získá z kombinace l její první hranu
19:            S := Push p to S
   ▷ Uloží hranu p do zásobníku pro výsledné spojení S
```

```
20:             c' := Get l without first edge
    ▷ Přiřadí do proměnné c' kombinaci l bez její první hrany
21:             if Get end node from combination c' = e then
    ▷ Pokud se uzel vycházející z poslední hrany kombinace c' rovná listu e
22:                 return S
    ▷ V zásobníku S se nachází spojení z kombinace c do listu e
23:             else
24:                 Repeat from line 6
    ▷ Spojení v zásobníku S nekončí v listu e, proces se opakuje
25:             end if
26:         end if
27:     end for
28: end for
```

Funkce getPathBeforeForDepthOne a getPathAfterForDepthOne

Rozdíl mezi getPathBeforeForDepthOne oproti getPathBefore a getPathAfterForDepthOne oproti getPathAfter je v Algoritmech 5 a 6 shodně na řádcích 1, 7, 10 a 20.

Na řádku 1 je místo odstranění poslední (Algoritmus 6), respektive první (Algoritmus 5) hrany z kombinace v parametru c, do proměnné c' uložena samotná kombinace.

Na řádku 7 je ve funkci getPathBeforeForDepthOne v podmínce zjištěno, zda se uzel l rovná počátečnímu uzlu první hrany spojení v proměnné c', ve funkci getPathAfterForDepthOne zda se uzel l rovná koncovému uzlu poslední hrany ve spojení v proměnné c'.

Na řádku 10 je do proměnné l' ve funkci getPathBeforeForDepthOne uložen výsledek funkce getAnotherBeforeForDepthOne(c, v), ve funkci getPathAfter(c, v) výsledek funkce getAnotherAfterForDepthOne.

Na řádku 20 se u obou funkcí opakuje situace z řádku č. 1, ovšem oproti řádku 1 neprobíhá odstranění první, respektive poslední hrany z kombinace c, ale z kombinace l.

Napojování kombinací na spojení

Pokud vytvářená spojení ve funkcích getPathBefore a getPathAfter ještě nevedou k cíli, hledají se kombinace k napojení. A pro nalezení nejvýhodnější takové kombinace se ve funkci getPathBefore používá funkce getAnotherBefore ve funkci getPathAfter funkce getAnotherAfter.

getAnotherBefore hledá nejlepší kombinaci, kterou lze napojit na spojení, jež má vést z uzlu start. getAnotherAfter oproti tomu hledá nejlepší kombinaci, kterou lze napojit na spojení, jež má vést do některého z listů grafu.

Obě zmíněné funkce fungují velmi podobně, a proto zde uvádím pouze pseudokód funkce getAnotherBefore (Algoritmus 7). Vstupními parametry jsou spojení c, na které se má navázat hledaná kombinace, a počet v, značící kolikrát byla původní kom-

binace (pro kterou se hledá alternativa) použita v nějakém scénáři. Pseudokód navíc obsahuje proměnnou T, symbolizující tabulku kombinací, a počítadlo průchodů V.

Princip funkce getAnotherBefore

(a) Projde každou kombinace u každého větvícího bodu.

(b) Odstraní první uzel z procházené kombinace a porovná ho s kombinací v parametru.

(c) Pokud se rovnají, tato se stává kandidátem na výhodnější kombinaci.

(d) Porovná se, zda je opravdu nová kombinace výhodnější, tedy méně krát použitá v testovacích scénářích, než ta původní.

(e) Nejméně krát použitá kombinace je vrácena (může se stát, že bude vrácen prázdný zásobník, což znamená že žádná výhodnější alternativa nebyla nalezena).

Algoritmus 7 getAnotherBefore(c, v, T, V)

```
1:  s := Init variable for resulting combination
    ▷ Inicializuje proměnnou pro výslednou kombinaci
2:  K := Get the keys from T
    ▷ Získá všechny klíče z tabulky kombinací T
3:  for k ∈ K do
    ▷ Projde všechny větvící body z tabulky kombinací
4:      L := Get the list of combinations for k from T
    ▷ Získá z tabulky T všechny kombinace u klíče k
5:      for l ∈ L do
    ▷ Pro jednotlivé kombinace
6:          if Get l without first edge = c then
    ▷ Pokud se kombinace l bez své první hrany rovná kombinaci c, lze ji napojit před c
7:              n := Get from V how many times was l visited
    ▷ Kolikrát byla kombinace použita v některém ze scénářů
8:              if n ≠ 0 then
    ▷ Pokud již byla použita
9:                  if n < v then
    ▷ Pokud byla nová kombinace méně krát použitá, než ta původní
10:                     s := l
    ▷ Uloží kombinaci jako tu aktuálně nejvýhodnější
11:                 end if
12:             else
    ▷ Pokud kombinace nebyla použita v žádném scénáři
```

```
13:                 return l
    ▷ Vrátí kombinaci l
14:             end if
15:         end if
16:     end for
17: end for
18: return s
    ▷ Vrátí nejvýhodnější kombinaci do spojení c
```

Princip funkce getAnotherAfter

(a) Projde každou kombinace u každého větvícího bodu.

(b) Odstraní poslední uzel z procházené kombinace a porovná ho s kombinací v parametru funkce.

(c) Pokud se rovnají a pokud existuje spojení do listu v parametru, tato se stává kandidátem na výhodnější kombinaci.

(d) Porovná se, zda je opravdu nová kombinace výhodnější, tedy méně krát použitá v testovacích scénářích, než ta původní.

(e) Nejméně krát použitá kombinace je vrácena (může se stát, že bude vrácen prázdný zásobník, což znamená že žádná výhodnější alternativa nebyla nalezena).

Rozdíl u pseudokódu funkce getAnotherAfter by byl zaprvé v přidání parametru e, tedy listu, do kterého má jít hledané spojení, a zadruhé na řádku 6, kde by se místo kombinace l bez první hrany porovnala s kombinací c kombinace l bez poslední hrany, a navíc by na té samé řádce došlo k porovnání, zda je možné z kombinace l dojít do listu e.

Funkce getAnotherBeforeForDepthOne a getAnotherAfterForDepthOne

Rozdíl u těchto funkcí je už v parametru, kde c není frontou hran, avšak pouhou jednou hranou. V případě funkce getAnotherBeforeForDepthOne by to znamenalo oproti pseudokódu Algoritmu 7 změnu na řádku č. 6, kde by místo porovnání kombinace c s kombinací l bez první hrany bylo zjištěno, zda se počáteční uzel hrany v parametru c rovná koncovému uzlu procházené kombinace l.

U funkce getAnotherAfterForDepthOne je to podobné, místo porovnání kombinace c s kombinací l bez poslední hrany, by se na řádku č. 6 porovnalo, zda se koncový uzel kombinace c rovná počátečnímu uzlu procházené kombinace l a zároveň zda je možné z kombinace l dojít do listu předaného v parametru této funkce.

5.3 PCT - PA

Algoritmus PCT - PA vytváří testovací scénáře podle metodiky Process Cycle Test (popsané v sekci 2.3.1) a jeho princip je téměř totožný s algoritmem PCT, popsaným v sekci 5.2. Oproti němu je však upraven tak, aby ve scénářích, pokud je to možné, byly použity hrany s vyšší prioritou.

5.3.1 Popis algoritmu

Stejně jako PCT, i algoritmus PCT - PA používá pro řízení generování testovacích scénářů funkci GENERATETESTSITUATIONSPCT (Algoritmus 1). Uvnitř ní se nejdříve funkcí CREATEALLCOMBINATIONSFORNODES (Algoritmus 2), vytvoří testovací kombinace a následně se z testovacích kombinací funkcí CREATETESTSITUATIONSPCT (Algoritmus 4) vytvoří testovací scénáře.

Uvnitř funkce CREATETESTSITUATIONSPCT se prochází všechny kombinace, které ještě nebyly použity v některém scénáři, a pro každou z nich je vždy vytvořené nové spojení. Toto spojení je složeno ze tří částí. První část vede z uzlu start do dané kombinace a hledá jej funkce GETPATHBEFORE (Algoritmus 5), druhá část je tvořena samotnou kombinací a třetí část spojení vede z kombinace do listu, který je uveden v parametru funkce CREATETESTSITUATIONSPCT, a hledá ho funkce GETPATHAFTER (Algoritmus 4).

Algoritmy PCT a PCT - PA se liší vlastně pouze ve výběru alternativní kombinace pro napojení na vytvářené spojení, tedy funkcemi GETANOTHERAFTER a GETANOTHERBEFORE. Zatímco v algoritmu PCT se spojení tvoří z kombinací s co nejmenším počtem průchodů, algoritmus PCT - PA se mimo to ještě snaží preferovat kombinace, které mají větší prioritu.

Porovnání priority kombinací je realizováno pomocí aritmetického průměru jejích hran. Převod textově reprezentované priority hrany na celá čísla je proveden následujícím způsobem - **nízká** („low“) priorita má váhu 1, **střední** („medium“) priorita má váhu 2 a **vysoká** („high“) priorita má váhu 3.

5.3.2 Implementační rozdíly oproti algoritmu PCT

Upřednostnění kombinací s větší prioritou hran je realizováno funkcemi GETANOTHERAFTERPA a GETANOTHERBEFOREPA, které jsou alternativou pro původní GETANOTHERAFTER, respektive GETANOTHERBEFORE (Algoritmus 7).

Funkce getAnotherAfterPA a getAnotherBeforePA

GETANOTHERAFTERPA, volaná z GETPATHAFTER, hledá nejlepší kombinaci, kterou lze napojit na první hranu kombinace v parametru, GETANOTHERBEFOREPA, volaná z GETPATHBEFORE, hledá nejlepší kombinaci, kterou lze napojit na poslední hranu kombinace v parametru.

Obě funkce jsou si opět velmi podobné, a tak přikládám pouze pseudokód funkce GETANOTHERBEFOREPA (číslo 8). Tento přijímá parametr c s kombinací, na kterou se má

napojit další kombinace, v s počtem kolikrát byla původní kombinace použita v nějakém scénáři a v pseudokódu navíc ještě tabulku kombinací T a počítadlo průchodů V.

Pseudokód funkce getAnotherAfterPA by se oproti Algoritmu 8 lišil jednak přidáním parametru e s listem grafu, do kterého se hledá spojení, a pak také řádkem č. 7, kde by se do proměnné c uložila kombinace l bez její poslední hrany a navíc by se zkontrolovalo, že lze z kombinace l dojít do listu e.

Algoritmus 8 getAnotherBeforePA(c, v, T, V)

```
1:  b := 0
    ▷ Inicializuje proměnou pro uložení arit. průměru priorit hran v nejlepší kombinaci
2:  s := Init variable for resulting combination
    ▷ Inicializuje proměnnou pro výslednou kombinaci
3:  K := Get the keys from T
    ▷ Získá všechny klíče z tabulky kombinací T
4:  for k ∈ K do
    ▷ Projde všechny větvící body z tabulky kombinací
5:      L := Get the list of combinations for k from T
    ▷ Získá z tabulky T všechny kombinace u klíče k
6:      for l ∈ L do
    ▷ Pro jednotlivé kombinace
7:          if Get l without first edge = c then
    ▷ Pokud se kombinace l bez své první hrany rovná kombinaci c, lze ji napojit před c
8:              n := Get from V how many times was l visited
    ▷ Kolikrát byla kombinace použita v některém ze scénářů
9:              if n ≠ 0 then
    ▷ Pokud již byla kombinace l použita v některém scénáři
10:                 b′ := 0
    ▷ Vytvoří proměnnou b′ pro počítání aritmetického průměru priorit hran v kombinaci l
11:                 for e ∈ l do
    ▷ Pro jednotlivé hrany v kombinaci
12:                     b′ := b′ + Get priority of e
    ▷ Do proměnné b′ přičte číselnou reprezentaci priority hrany e
13:                 end for
14:                 b′ := b′ ÷ Number of edges in l
    ▷ Vydělí proměnnou b′ počtem hran v kombinaci l
15:                 if b ≠ 0 then
    ▷ Pokud již byla nalezena nějaká kombinace
16:                     if n < v then
    ▷ Pokud má nově nalezená alternativa menší počet návštěv, než původní varianta
```

```
17:                         m := m′
18:                         s := l
    ▷ Průměr hran m′ a kombinace l jsou nově těmi nejvýhodnějšími
19:                     else if B < P then
    ▷ Nově nalezená alternativa má větší průměrnou prioritu
20:                         m := m′
21:                         s := l
    ▷ Průměr hran m′ a kombinace l jsou nově těmi nejvýhodnějšími
22:                     end if
23:                 else
    ▷ Pokud ještě nebyla nalezena žádná kombinace
24:                     m := m′
25:                     s := l
    ▷ Kombinace l a průměr m′ jsou těmi nejvýhodnějšími
26:                 end if
27:             else
    ▷ Kombinace l ještě nebyla použita ve scénáři
28:                 return l
    ▷ Vrátí nejvýhodnější kombinaci l
29:             end if
30:         end if
31:     end for
32: end for
33: return s
    ▷ Vrátí nejvýhodnější kombinaci
```

GetAnotherBeforeForDepthOnePA a getAnotherAfterForDepthOnePA

Pseudokód funkce getAnotherBeforeForDepthOnePA, tedy té, která hledá alternativní kombinaci pro spojení z uzlu start do kombinace při hloubce pokrytí rovné jedné, by se oproti funkci getAnotherBeforePA (Algoritmus 8) lišil pouze řádkem č. 7, kde místo porovnání kombinace c s kombinací l bez poslední hrany, by se porovnalo, zda se počáteční uzel kombinace c rovná koncovému uzlu procházené kombinace l.

Pseudokód funkce getAnotherAfterForDepthOnePA, tedy té, která hledá alternativní kombinaci pro spojení z kombinace do listu v parametru při hloubce pokrytí rovné jedné, by se oproti pseudokódu funkce getAnotherAfterPA taktéž lišil pouze řádkem s porovnáním kombinace c a kombinace l (tentokrát bez první hrany), kdy by místo toho došlo k porovnání, zda se koncový uzel kombinace c rovná počátečnímu uzlu procházené kombinace l a zároveň zda je možné z kombinace l dojít do listu předaného v parametru této funkce.

5.4 PCT SmokeTest

V této sekci popisuji algoritmus PCT SmokeTest, který realizuje stejnojmennou techniku pro získání testovacích scénářů, definovanou v úvodu v sekci 2.3.2. Tento algoritmus jsem vytvořil společně s vedoucím mé práce a jde jím o významný přínos této práce. Algoritmus PCT SmokeTest je optimalizován na generování co nejmenšího počtu hran ve scénářích, tedy aby bylo testování aplikace co nejrychlejší.

5.4.1 Popis algoritmu

(a) Nejprve je vytvořena množina PRIO (její definice je uvedena sekci 2.3.2).

(b) Poté jsou pro každý prvek z množiny PRIO nalezeny všechny listy, do kterých existuje nějaké spojení. Nalezené listy označím jako prvky množiny LEAFS.

(c) Pro každý list z množiny LEAFS jsou následně hledána všechna spojení, která vedou z uzlu start do tohoto listu a která obsahují alespoň jednu kombinaci z množiny PRIO. Tato spojení označím jako prvky množiny LOCAL_PATHS.

(d) Z množiny LOCAL_PATHS je nalezeno nejlepší spojení (obsahující nejvíc kombinací z množiny PRIO).

(e) Z množiny PRIO jsou vyškrtnuty ty kombinace, které jsou obsaženy v nejlepším spojení.

(f) Dokud není množina PRIO prázdná, proces se opakuje.

5.4.2 Řídící funkce

Generování testovacích scénářů řídí jediná funkce GENERATETESTSITUATIONSST (Algoritmus 9) ze třídy `PCTSmokeTestMainFunctions`. Tato funkce přijímá parametr s modelem grafu m, hloubku pokrytí d a hodnotu STL t a volá několik funkcí ze třídy `PCTSmokeTestSituationsGenerator`. Její princip je následující:

(a) První krok generování testovacích scénářů tímto algoritmem je stejný jako v případě algoritmů PCT a PCT - PA. Tedy až do odpovídající hloubky pokrytí (TDL), jsou funkcí CREATEALLCOMBINATIONSFORNODES (Algoritmus 2) nalezeny všechny kombinace (tento proces popisuji již v sekci 5.2.3).

(b) Z těchto kombinací a některých dalších hran jsou vybrány funkcí CREATESMOKECOMBINATIONS (Algoritmus 10) ty, jež vyhovují podmínkám definovaným v sekci 2.3.2 (těmto vyhovujícím kombinacím a hranám říkám prvky množiny PRIO).

(c) Ze seznamu kombinací odpovídajícím podmínkám množiny PRIO je vytvořena tabulka kombinací, ve které je klíčem první uzel v dané kombinaci a ve které jsou hrany v jednotlivých kombinacích přehozeny do opačného pořadí (stejně jako v sekci 5.2.3).

(d) Nakonec je zavolána funkce `createTestSituationsST` (Algoritmus 11), která z PRIO kombinací v tabulce vytvoří testovací scénáře.

Algoritmus 9 generateTestSituationsST(m, d, t)

1: R := Init list for resulting situations
▷ Inicializuje seznam pro výsledné testovací scénáře
2: N := All nodes in graph m
▷ Do proměnné N uloží všechny uzly z grafu
3: s := Get start node of graph m
▷ Uloží do proměnné s uzel start
4: C := createAllCombinationsForNodes(d, N)
▷ Zavolá funkci, která najde testovací kombinace, a ty se uloží do proměnné C
5: S := createSmokeCombinations(t, d, C, m)
▷ Vytvoří seznam kombinací splňujících podmínky 2.3.2
6: **for** $s \in S$ **do**
▷ Projde kombinace ze seznamu S
7: T := Add s to table T
▷ Každou kombinaci s zařadí podle koncového uzlu první hrany do tabulky T
8: **end for**
9: R := createTestSituationsST(s, T)
▷ Zavolá funkci, která najde testovací scénáře
10: **return** R
▷ Vrátí seznam testovacích scénářů

5.4.3 Vytvoření množiny PRIO

Množina PRIO je vytvářena funkcí createSmokeCombinations (Algoritmus 10). Tato funkce přijímá parametr se STL t, hloubkou pokrytí d a v pseudokódu ještě seznam kombinací C, reprezentovaných v kódu statickou proměnnou *combinations*, a model grafu m. Výsledkem běhu této funkce je v pseudokódu proměnná R, v implementaci statická proměnná *smokeCombinations* s kombinacemi, jež odpovídají definici 2.3.2. Toho se docílí následujícími kroky:

(a) Nejprve se projdou všechny testovací kombinace vytvořené funkcí createAllCombinationsForNodes (Algoritmus 2). Ty, které mají první hranu s nejvyšší prioritou (v případě STL nastaveného na „medium“ i se střední prioritou), jsou zkopírovány do výsledné proměnné.

(b) V případě, že je TDL větší než 1 následuje další cyklus, který projde všechny hrany v grafu. Pokud procházená hrana není ve výsledném seznamu, a je prioritní (opět u STL „high“ má nejvyšší, u STL „medium“ i střední prioritu), je také přidána do výsledného seznamu.

Algoritmus 10 createSmokeCombinations(t, d, C, m)

```
1: R := Init variable for resulting list of PRIO combinations
   ▷ Vytvoří proměnnou R, do které se budou ukládat nalezené PRIO kombinace
2: for c ∈ C do
   ▷ Projde kombinace ve vstupním parametru C
3:     e := Get last edge in c
   ▷ Získá z kombinace první hranu
4:     p := Get priority of e
   ▷ Získá prioritu první hrany v kombinaci
5:     if p = 'HIGH' then
   ▷ Pokud má první hrana kombinace nejvyšší prioritu
6:         R := Add combination c to R
   ▷ Přidá kombinaci c do seznamu kombinací R
7:     end if
8:     if t = 'MEDIUM' then
   ▷ Pokud je STL nastaven na „Medium“
9:         if p = 'MEDIUM' then
   ▷ Pokud má první hrana kombinace střední prioritu
10:            R := Add combination c to R
   ▷ Přidá kombinaci c do seznamu kombinací R
11:        end if
12:    end if
13: end for
14: if d > 1 then
   ▷ Pokud je hloubka pokrytí větší než jedna, musí se zkontrolovat, zda v R nechybí
   prioritní kombinace s délkou 1
15:    for e ∈ Get all edges in m do
   ▷ Pro každou hranu v grafu
16:        if e is not contained in R then
   ▷ Pokud hrana e není obsažena v některé z kombinací v R
17:            p := Get priority of e
   ▷ Získá prioritu hrany e
18:            if p = 'HIGH' then
   ▷ Pokud má hrana e nejvyšší prioritu
19:                R := Add edge e to R
   ▷ Přidá hranu e do seznamu kombinací R
20:            end if
21:            if t = 'MEDIUM' then
   ▷ Pokud je STL nastaven na „Medium“
22:                if p = 'MEDIUM' then
   ▷ Pokud má hrana e střední prioritu
23:                    R := Add edge e to R
   ▷ Přidá hranu e do seznamu kombinací R
```

```
24:                 end if
25:             end if
26:         end if
27:     end for
28: end if
29: return R
    ▷ Vrátí seznam kombinací R
```

5.4.4 Funkce pro generování testovacích scénářů

Generování testovacích scénářů řídí funkce createTestSituationsST (Algoritmus 11). Funkce má v parametru uveden uzel start s a tabulku kombinací T (v implementaci reprezentovanou statickou proměnnou *smokeCombinationsTable*) a jejím výsledkem je naplnění proměnné R (v implementaci statické proměnné *testSituations*) hotovými testovacími scénáři. Princip funkce je následující:

(a) Pro každou kombinaci z tabulky T, jsou funkcí findEnds (Algoritmus 12) nalezeny všechny listy, do kterých lze z kombinace dojít (množina LEAFS).

(b) Pro každý prvek z množiny LEAFS jsou následně funkcí getAllPathsToStart (Algoritmus 13) hledána všechna spojení, která vedou z uzlu start s a která obsahují alespoň jednu kombinaci z množiny PRIO.

(c) Když jsou nalezena spojení pro všechny prvky množiny LEAFS, tedy je naplněna kompletní množina ALL_PATHS, hledá se funkcí selectMostExpensivePath (Algoritmus 14) nejlepší spojení.

(d) Nejlepší spojení, nalezené v předchozím kroku, je vyhodnoceno jako testovací scénář a přidáno do vytvářeného seznamu scénářů.

(e) Z tabulky T se odstraní ty kombinace, které jsou obsaženy v nejlepším spojení.

(f) Pokud v tabulce T zbývají ještě nějaké scénáře, proces se opakuje.

Algoritmus 11 createTestSituationsST(s, T)

```
1: do
2:     R := Init list for resulting situations
   ▷ Inicializuje seznam pro výsledné testovací scénáře
3:     P := Init list for all founded paths
   ▷ Inicializuje seznam pro všechna nalezená spojení z uzlu start do některého z listů
4:     K := Get the keys from T
   ▷ Získá z tabulky kombinací T všechny klíče
5:     for k ∈ K do
   ▷ Projde všechny větvící body z tabulky kombinací
```

6: C := GET THE LIST OF COMBINATIONS FOR k FROM T
▷ Získá z tabulky T všechny kombinace u klíče k
7: **for** $c \in C$ **do**
▷ Projde všechny kombinace od větvícího bodu
8: c' := GET THE LAST NODE IN c
▷ Najde uzel, do kterého vstupuje poslední hrana v kombinaci
9: E := INIT LIST FOR STORING LEAFS
▷ Vytvoří seznam pro listy dosažitelné z uzlu s
10: Q := INIT EMPTY QUEUE
▷ Vytvoří frontu pro funkci findEnds
11: V := INIT EMPTY MAP
▷ Vytvoří mapu pro počítadlo průchodů, ve které bude klíčem hrana a hodnotou celé číslo
12: E := findEnds(c', Q, E, V)
▷ Volá funkci pro nalezení všech listů od uzlu s, které uloží do E
13: **for** $e \in E$ **do**
▷ Pro každý list z E
14: V := INIT EMPTY MAP
▷ Vytvoří mapu pro počítadlo průchodů, ve které bude klíčem hrana a hodnotou celé číslo
15: S := INIT EMPTY STACK
▷ Vytvoří nový zásobník
16: P := getAllPathsToStart(e, s, S, P, T, V)
▷ Volá funkci pro nalezení všech spojení z listu do uzlu start, které uloží do P
17: **end for**
18: **end for**
19: **end for**
20: B := selectMostExpensivePath(P)
▷ Volá funkci pro nalezení nejlepšího spojení ze seznamu P
21: R := ADD B TO R
▷ Přidá B do seznamu testovacích scénářů R
22: **for** $k \in K$ **do**
▷ Projde všechny větvící body z tabulky kombinací
23: U := INIT LIST FOR UNEXPLORED COMBINATIONS
▷ Vytvoří seznam pro nepoužité kombinace
24: C := GET THE LIST OF COMBINATIONS FOR k FROM T
▷ Vrátí z tabulky T všechny kombinace u klíče k
25: **for** $c \in C$ **do**
▷ Projde všechny kombinace od větvícího bodu
26: o = DOES B CONTAIN COMBINATION c
▷ Do o uloží *true*, pokud spojení B obsahuje kombinaci c

27: **if not** o **then**
▷ Pokud kombinace c není ve spojení obsažena
28: ADD c TO U
▷ Přidá kombinaci do seznamu neprošlých kombinací
29: **end if**
30: **end for**
31: **if** GET THE SIZE OF $U = 0$ **then**
▷ Pokud je seznam neprošlých kombinací prázdný
32: REMOVE k FROM T
▷ Z tabulky T může být odstraněn klíč k, neboť všechny u něj uložené kombinace jsou již obsažené v nalezených scénářích
33: **else**
▷ Pokud seznam neprošlých kombinací není prázdný
34: REPLACE CURRENT VALUES FOR k IN T WITH U
▷ V tabulce T se nahradí u klíče k původní hodnoty za kombinace v U
35: **end if**
36: **end for**
37: **while** GET ALL VALUES FROM $T \neq 0$
▷ Cyklus se opakuje dokud tabulka T obsahuje nějaké kombinace
38: **return** R
▷ Funkce vrátí seznam s nalezenými testovacími scénáři R

5.4.5 Funkce pro hledání všech listů z uzlu

Pro nalezení všech listů, dosažitelných z nějakého uzlu, jsem vytvořil funkci FINDENDS (Algoritmus 12). Tato funkce ke své činnosti potřebuje dva parametry - uzel n, ze kterého se listy hledají a frontu Q s právě procházenými hranami. Pseudokód navíc obsahuje ještě parametr E, do kterého jsou výsledné listy ukládány (v implementaci reprezentovaném statickou proměnnou *combinationsEnds*) a počítadlo průchodů V (v implementaci statická proměnná $visitedCounterMap$). Princip této funkce je následující:

(a) Z uzlu n, pro který se hledají listy, se vyberou všechny neprošlé výstupní hrany.

(b) Pokud pro uzel n žádná taková hrana neexistuje, uzel je listem a je přidán do výsledného seznamu listů LEAFS (v pseudokódu proměnná E).

(c) Pro každou neprošlou hranu, vystupující z uzlu n, je do počítadla průchodů V započten jeden průchod a její koncový uzel je přidán do fronty zbývajících hran (v pseudokódu proměnná Q).

(d) Pokud fronta zbývajících hran není prázdná, rekurzivně se zavolá tatáž funkce, ve které je uzlem, pro který se hledají listy, první prvek z fronty Q. Ostatní vstupní proměnné zůstanou stejné (fronta Q a v pseudokódu seznam listů E).

Algoritmus 12 findEnds(n, Q, E, V)

```
1: O := Init list for outgoing edges
   ▷ Inicializuje seznam pro hrany s počátečním uzlem n
2: U := Init list for unvisited outgoing edges
   ▷ Inicializuje seznam pro neprošlé hrany s počátečním uzlem n
3: for e ∈ Get all outgoing edges from n do
   ▷ Projde hrany vystupující z n
4:     O := Add e to O
   ▷ Přidá hranu do seznamu
5:     v := Get from V how many times was e visited
   ▷ Kolikrát byla hrana procházena
6:     if v = 0 then
   ▷ Pokud hrana ještě nebyla procházena
7:         U := Add e to U
   ▷ Přidá hranu do seznamu neprošlých hran
8:     end if
9: end for
10: if Get number of edges in O = 0 and Get number of edges in U = 0 then
   ▷ Pokud z uzlu n nevystupuje žádná hrana
11:     E := Add n to list of leafs E
   ▷ Přidá nalezený uzel do seznamu listů E
12: else
   ▷ Z uzlu n vystupuje nějaká hrana
13:     for e ∈ U do
   ▷ Projde všechny neprošlé výstupní hrany
14:         v := Get from V how many times was e visited
   ▷ Kolikrát byla hrana procházena
15:         Add a visit of e to V
   ▷ Započítá průchod hranou
16:         t := Get target of e
   ▷ Vrátí koncový uzel dané hrany
17:         Append t to Q
   ▷ Přidá uzel do fronty
18:     end for
19: end if
20: if Get the number of elements in Q ≠ 0 then
   ▷ Pokud ve frontě ještě zbývá nějaký prvek
21:     f := Poll first element from Q
   ▷ Odebere a uloží do nové proměnné první položku z fronty
```

```
22:     findEnds(f, Q, E, V)
    ▷ Rekurzivní volání
23: else
    ▷ Pokud je fronta prázdná
24:     return E
    ▷ Funkce vrátí seznam s nalezenými listy
25: end if
```

5.4.6 Funkce pro nalezení všech spojení ze startu do listu

Pro hledání všech spojení v grafu mezi uzlem start a některým z listů jsem vytvořil funkci getAllPathsToStart (Algoritmus 13). Jejími vstupními parametry jsou: uzel n, do kterého má spojení vést, uzel start r a zásobník s aktuálně procházeným spojením S. V pseudokódu této funkce uvádím jako vstupní parametr ještě zásobník s nalezenými spojeními R (v implementaci statická proměnná *pathsToStart*), tabulku kombinací T (v implementaci statická proměnná *smokeCombinationsTable*) a počítadlo průchodů V (v implementaci statická proměnná $visitedCounterMap$).

Princip funkce getAllPathsToStart

(a) Vyberou se všechny maximálně jedenkrát prošlé hrany, jejíž cílovým uzlem je uzel v parametru (v pseudokódu Algoritmus 13 označen písmenem n).

(b) Projdou se všechny hrany nalezené v předchozím kroku.

(c) Každá hrana je přidána na zásobník a je jí přičten průchod. Následně se zkontroluje, zda uzel s na jejím začátku není uzlem start.

   (a) Pokud je, a pokud se v zásobníku vyskytuje některá z kombinací z množiny PRIO, z obsahu zásobníku je vytvořeno nové spojení, které je přidáno do cílového seznamu. Dále se zkontroluje, zda z uzlu s vychází nějaká neprošlá hrana. Pokud ano, rekurzivně se zavolá tatáž funkce, ovšem počátečním uzlem (v pseudokódu 13 parametr n) je nově uzel s. Pokud ne, ze zásobníku se odstraní první prvek (hrana) a tomu se odečte z počítadla průchodů jeden průchod.

   (b) Pokud není, rekurzivně se zavolá tatáž funkce, ovšem počátečním uzlem (v pseudokódu Algoritmus 13 parametr n) je nově uzel s.

(d) Pokud zásobník S není prázdný, je z něj odstraněna první hrana. Zároveň je této hraně z počítadla průchodů odečten jeden průchod.

Algoritmus 13 getAllPathsToStart(n, r, S, R, T, V)

```
1: I := Init list for incoming edges to n
   ▷ Inicializuje seznam pro hrany s koncovým uzlem n
2: for e ∈ Get all incoming edges to n do
   ▷ Projde hrany s koncovým uzlem n
3:     v := Get from V how many times was e visited
   ▷ Kolikrát byla hrana procházena
4:     if v < 2 then
   ▷ Pokud byla hrana procházena méně než dvakrát
5:         I := Add e to I
   ▷ Přidá hranu do seznamu I
6:     end if
7: end for
8: for e ∈ I do
9:     s := Get source of e
   ▷ Vrátí počáteční uzel hrany e
10:    S := Push e to S
   ▷ Přidá hranu do zásobníku
11:    Add a visit of e to V
   ▷ Započítá průchod hranou
12:    if s = r then
   ▷ Pokud je s uzlem start
13:        if S contains any combination from T then
   ▷ Pokud zásobník obsahuje nějakou kombinaci z tabulka kombinací T
14:            R := Add a copy of S to list R
   ▷ Obsah zásobníku je přidán do seznamu nalezených spojení
15:        end if
16:        if Get all edges incoming from s ≠ 0 then
   ▷ Pokud je počet hran vystupujících z s větší než 0
17:            getAllPathsToStart(s, r, S, R, T, V)
   ▷ Rekurzivní volání
18:        else
19:            f := Pop first element from S
   ▷ Odebere a uloží do nové proměnné první položku ze zásobníku
20:            Add a visit of f to V
   ▷ Započítá průchod hranou
21:        end if
22:    else
23:        getAllPathsToStart(s, r, S, R, T, V)
   ▷ Rekurzivní volání
24:    end if
25: end for
```

26: **if** Get the number of elements in $S \neq 0$ **then**
▷ Pokud v zásobníku ještě zbývá nějaký prvek
27: $f :=$ Pop first element from S
▷ Odebere a uloží do nové proměnné první položku ze zásobníku
28: Remove a visit of e from V
▷ Z počítadla průchodů odečte jedničku
29: **else**
30: **return** R
▷ Pokud je zásobník prázdný, vrátí seznam nalezených cest
31: **end if**

5.4.7 Funkce pro nalezení nejvýhodnějšího spojení

Pro nalezení nejvýhodnějšího spojení ze seznamu spojení z uzlu start do listů jsem vytvořil funkci selectMostExpensivePath (Algoritmus 14). Směrodatným kritériem pro porovnání spojení je obsažený počet kombinací z množiny PRIO. Tedy čím více kombinací z množiny PRIO spojení obsahuje, tím je výhodnější. V případě shodného počtu kombinací je vybráno kratší spojení (s menším počtem hran).

Funkce `selectMostExpensivePath(P)` přijímá pouze parametr P se seznamem spojení k porovnání.

Algoritmus 14 selectMostExpensivePath(P)

1: $B :=$ Init variable for best path
▷ Inicializuje proměnnou pro nejlepší spojení
2: $n :=$ Init variable for storing the number of PRIO combinations in B
▷ Inicializuje proměnnou pro uložení počtu PRIO kombinací v nejlepším spojení
3: **for** $p \in P$ **do**
▷ Projde všechna spojení
4: $n' :=$ Count PRIO combinations in p
▷ Zjistí kolik cesta obsahuje kombinací z množiny PRIO
5: **if** $B = null$ **then**
▷ Pokud ještě nebylo nalezeno žádné spojení
6: $n := n'$
7: $B := p$
▷ Uloží procházené spojení p a počet kombinací n' jako ty nejvýhodnější
8: **else if** $n < n'$ **then**
▷ Pokud je v novém spojení více kombinací z množiny PRIO, než ve starém spojení
9: $n := n'$
10: $B := p$
▷ Uloží procházené spojení p a počet kombinací n' jako ty nejvýhodnější

11: **else if** $n = n'$ **and** NUMBER OF EDGES IN $B >$ NUMBER OF EDGES IN p **then**
▷ Pokud je počet kombinací z množiny PRIO v obou spojeních stejné, a pokud je počet hran v novém spojení menší
12: $n := n'$
13: $B := p$
▷ Uloží procházené spojení p a počet kombinací n' jako ty nejvýhodnější
14: **end if**
15: **end for**
16: **return** B
▷ Vrátí nejvýhodnější spojení

5.5 PCT SmokeTest II

Při tvorbě algoritmu PCT SmokeTest (o kterém píši v části 5.4), který automatizuje stejnojmennou techniku na generování testovacích scénářů (vysvětlenou v části 2.3.2), vznikla jeho alternativa. Pojmenovali jsme ji PCT SmokeTest II a její odlišnost od algoritmu PCT SmokeTest spočívá v maximalizaci počtu prioritních akcí (hran s prioritou „high“, nebo „medium“) v testovacích scénářích.

5.5.1 Popis algoritmu

Struktura tohoto algoritmu je totožná se strukturou algoritmu PCT SmokeTest. Vše řídí funkce GENERATETESTSITUATIONSST (Algoritmus 9), a jednotlivé kroky jsou tyto:

(a) Funkcí CREATEALLCOMBINATIONSFORNODES (Algoritmus 2) jsou nalezeny všechny kombinace (tento proces popisuji v sekci 5.2.3).

(b) Funkcí CREATESMOKECOMBINATIONS (Algoritmus 10) je vytvořena množina PRIO (definovanou v sekci 2.3.2).

(c) Pro každý prvek z množiny PRIO jsou funkcí FINDENDS (Algoritmus 12) nalezeny všechny listy, do kterých lze z kombinace dojít (nalezené listy označuji jako prvky množiny LEAFS).

(d) Pro každý list z množiny LEAFS jsou funkcí GETALLPATHSTOSTART (Algoritmus 13) nalezena všechna spojení, která vedou z uzlu start do tohoto listu (spojení označuji jako prvky množiny LOCAL_PATHS).

(e) Z množiny LOCAL_PATHS je funkcí SELECTMOSTEXPENSIVEPATHII (Algoritmus 15) nalezeno nejlepší spojení.

(f) Z množiny PRIO jsou vyškrtnuty ty kombinace, které jsou obsaženy v nejlepším spojení.

(g) Dokud není množina PRIO prázdná, proces se opakuje.

5.5.2 Implementační rozdíly oproti algoritmu PCT SmokeTest

Jediný rozdíl oproti algoritmu PCT SmokeTest je ve volání funkce selectMostExpensivePathII, oproti původní selectMostExpensivePath (Algoritmus 14), která hledá nejvýhodnější spojení ze seznamu spojení.

Funkce `selectMostExpensivePathII(P)` (Algoritmus 15) hledá nejvýhodnější položku ze seznamu spojení, které vedou z uzlu start do některého z listů. Nejdůležitějším kritériem pro porovnání spojení je, stejně jako u funkce selectMostExpensivePath (Algoritmus 14), počet kombinací z množiny PRIO. Ovšem pokud porovnávaná spojení obsahují těchto kombinací stejný počet, vybere se to, které má větší počet hran s nejvyšší prioritou. V případě, že i toto číslo je u obou spojení shodné, rozhodne počet hran se střední prioritou.

Algoritmus 15 selectMostExpensivePathII(P)

1: B := Init variable for best path
▷ Inicializuje proměnnou pro nejlepší spojení
2: n := Init variable for storing the number of PRIO combinations in B
▷ Inicializuje proměnnou pro uložení počtu PRIO kombinací v nejlepším spojení
3: M := Init map with keys of type String and values of type Integer
▷ Inicializuje proměnnou pro uložení mapy s množstvím priorit v nejlepším spojení
4: **for** $p \in P$ **do**
▷ Projde všechna spojení
5: N := Init map with keys of type String and values of type Integer
▷ Inicializuje proměnnou pro uložení mapy s počtem jednotlivých priorit v právě procházeném spojení
6: **for** $e \in p$ **do**
▷ Pro jednotlivé hrany ve spojení p
7: N := Count priority of e
▷ Do mapy započítá pod klíč s hodnotou priority hrany e jeden výskyt
8: **end for**
9: n' := Count PRIO combinations in p
▷ Zjistí kolik cesta obsahuje kombinací z množiny PRIO
10: **if** $B = null$ **then**
▷ Pokud ještě nebylo nalezeno žádné spojení
11: $n := n'$
12: $B := p$
13: $M := N$
▷ Uloží procházené spojení p, počet kombinací n' a mapu N jako ty nejvýhodnější
14: **else if** $n < n'$ **then**
▷ Pokud je v novém spojení více kombinací z množiny PRIO, než ve starém spojení
15: $n := n'$
16: $B := p$

17: $M := N$
▷ Uloží procházené spojení p, počet kombinací n' a mapu N jako ty nejvýhodnější
18: **else if** $n = n'$ **then**
▷ Pokud je v novém spojení stejně kombinací z množiny PRIO, jako v tom starém
19: $NH :=$ Get number of edges with high priority from N
▷ Zjistí z mapy počet hran s nejvyšší prioritou v procházeném spojení
20: $NM :=$ Count edges with medium priority in N
▷ Zjistí z mapy počet hran se střední prioritou v procházeném spojení
21: $MH :=$ Count edges with high priority in M
▷ Zjistí z mapy počet hran s nejvyšší prioritou v nejvýhodnějším spojení
22: $MM :=$ Count edges with medium priority in M
▷ Zjistí z mapy počet hran se střední prioritou v nejvýhodnějším spojení
23: **if** $MH < NH$ **then**
▷ Pokud je v nejvýhodnějším spojení méně hran s nejvyšší prioritou, než v procházeném spojení
24: $n := n'$
25: $B := p$
26: $M := N$
▷ Uloží procházené spojení p, počet kombinací n' a mapu N jako ty nejvýhodnější
27: **else if** $MH = NH$ **and** $MM < NM$ **then**
▷ Pokud je v nejvýhodnějším spojení stejně hran s nejvyšší prioritou, jako v procházeném spojení, ale je v něm méně hran se střední prioritou
28: $n := n'$
29: $B := p$
30: $M := N$
▷ Uloží procházené spojení p, počet kombinací n' a mapu N jako ty nejvýhodnější
31: **end if**
32: **end if**
33: **end for**
34: **return** B
▷ Vrátí nejvýhodnější spojení

5.6 Složitost algoritmů

Složitost implementovaných algoritmů je zobrazena v tabulce 5.1. Jednotlivé parametry značí: N počet uzlů v grafu, d hloubku pokrytí a K počet vygenerovaných kombinací.

Přestože se v nejhorším případě jedná o velmi neoptimální algoritmy, v průchodech reálnými grafy jsou algoritmy v průměrných případech o několik řádů rychlejší. To je vidět z tabulky A.7, která obsahuje čas trvání generování testovacích scénářů jednotlivými algoritmy u 20 testovaných grafů a ve které žádná z výsledných hodnot nedosahuje ani jedné vteřiny.

Algoritmus	**Části algoritmu**	**Složitost**	**Asymptotická složitost**
PCT	createAllCombinationsForNodes	$O(N \times O^d)$	$O(K^4)$
	createTestSituations	$O(K^4)$	
PCT - PA	createAllCombinationsForNodes	$O(N \times O^d)$	$O(d \times K^4)$
	createTestSituationsPA	$O(d \times K^4)$	
PCT SmokeTest	createAllCombinationsForNodes	$O(N \times O^d)$	$O(d \times K^4)$
	createSmokeCombinations	$O(N \times K)$	
	createTestSituationsST	$O(d \times K^4)$	
PCT SmokeTest II	createAllCombinationsForNodes	$O(N \times O^d)$	$O(d \times K^4)$
	createSmokeCombinations	$O(N \times K)$	
	createTestSituationsSTII	$O(d \times K^4)$	

Tabulka 5.1: *Asymptotická složitost jednotlivých algoritmů a jejich dílčích částí*

Kapitola 6

Testování

V této kapitole popisuji jak probíhalo testování aplikace a jednotlivých algoritmů. V první části mluvím o testování funkčnosti algoritmů, tedy že algoritmy generují testovací scénáře v souladu s definicí, ve druhé části porovnávám na vygenerovaných scénářích efektivitu algoritmů.

6.1 Test správnosti generování testovacích scénářů

Správnost generování testovacích scénářů jsem ověřoval s pomocí studentů předmětu TS1, který vyučuje vedoucí mé bakalářské práce. V rámci testování jsme ručně jednotlivými metodami vytvořili testovací scénáře a tyto jsem následně porovnal s těmi vygenerovanými pomocí mé aplikace. Porovnání výsledků několikrát odhalilo v automatickém generování závažnou chybu, po jejíž opravě se vždy změnila podoba výsledných scénářů, takže bylo potřeba provést automatické generování opakovaně.

Kromě jediného případu, a to u generování scénářů pomocí algoritmu PCT u grafu č. 12, vyprodukovaly všechny implementované algoritmy při posledním testování validní výsledky, které je možné vidět v tabulkách A.3 a A.4 (v příloze).

6.1.1 Specifikace testu

Kritérii, podle kterých jsem hodnotil správnost generování, byly počet testovacích scénářů a celkový počet hran ve scénářích. Nešlo totiž jen vzít scénáře vytvořené ručně a porovnat je s těmi automaticky generovanými, neboť dva různé výsledky generování testovacích scénářů pro jeden graf, algoritmus a hloubku pokrytí, nemusí být složeny ze stejných hran, a přesto mohou být v souladu se specifikací dané metody.

Testování proběhlo na čtyřech grafech, jejichž vlastnosti jsou uvedeny v tabulce A.1. Na dvou z nich (jednom obsahujícím cykly, druhém acyklickém) jsme testovali všechny algoritmy, na třetím jsme testovali algoritmy PCT a PCT - PA, a na posledním PCT SmokeTest a PCT SmokeTest II. Poslední dva grafy byly proto vybrány tak, aby se na nich dala co nejlépe otestovat správnost implementace dané metodiky.

Každý algoritmus jsme otestovali dvakrát, jednou s hloubkou pokrytí rovnou jedné, podruhé s hloubkou pokrytí rovné dvěma. Pro scénáře generované metodou PCT SmokeTest byl STL zvolen na „high“.

6.1.2 Hodnocení výsledku

Výsledek automatického generování testovacích scénářů jednotlivými algoritmy v jejich finální podobě je uveden v tabulce A.4, výsledek ručního generování scénářů je v tabulce A.3. Ve všech případech, ve kterých se výsledky liší, proběhla ruční kontrola správnosti automatického generování a ta žádné další chyby neodhalila.

Největší shoda mezi ručním a automatickým generováním byla u algoritmu PCT SmokeTest, který generoval stejný výsledek v 91 % případů. Při kontrole tohoto rozdílu jsem odhalil chybu v ručním generování, takže je u algoritmu PCT SmokeTest shoda s ručním generováním dokonce 100%.

Díky testu správnosti generování bylo odhaleno několik chyb. Uvedu zde dvě nejzásadnější.

- Testovací scénáře generované algoritmem PCT SmokeTest v určitých grafech s cyklem nezačínaly v uzlu start. Tato chyba byla ve funkci getAllPathsToStart (Algoritmus 13) opravena přidáním možnosti projít při tvorbě spojení každou hranu dvakrát, nejen jednou.

- Algoritmy implementující metodu PCT SmokeTest nezahrnovaly určité hrany do množiny PRIO. A to hrany, do jejichž počátečního uzlu nevede hrana s nejvyšší prioritou (případně při STL nastaveného na „medium“ i hrana se střední prioritou) a jejichž koncovým uzlem je list. Tyto hrany by při hloubce pokrytí větší než 1 mohly zůstat neotestované, což by bylo v rozporu s definicí metody PCT SmokeTest. Tato chyba byla také opravena, jak je popsáno v části 2.3.2.

6.2 Porovnání algoritmů

V této části shrnu rozdíly ve vygenerovaných testovacích scénářích jednotlivými algoritmy. Zadání, jak mají scénáře vypadat pro generování algoritmy PCT a PCT - PA, se liší od toho pro generovaných algoritmy PCT SmokeTest a PCT SmokeTest II[1], a proto vůči sobě porovnávám vždy jen dané implementace[2].

[1] V případě algoritmů implementujících techniku PCT se musí ve scénářích použít všechny testovací kombinace, zatímco u algoritmů implementujících techniku PCT SmokeTest se musí použít jen ty, vyhovující definici uvedené v sekci 2.3.2.

[2] Kdybych vzájemně porovnával všechny čtyři algoritmy, byl by výsledek jednoznačně ve prospěch algoritmu PCT SmokeTest - například průměrný počet hran ve scénářích je u algoritmu PCT 29, u PCT - PA 30, u PCT SmokeTest II 17 a u PCT SmokeTest pouhých 13 hran.

6.2.1 Specifikace testu

Na 20 grafech nahraných do aplikace (vlastnosti grafů uvedeny v tabulce A.2) jsem porovnával následující kritéria: počet scénářů (výsledek v tabulce A.5), počet hran ve scénářích (výsledek v tabulce A.6), priority hran ve scénářích (výsledek v tabulce A.7) a časovou náročnost generování testovacích scénářů (výsledek v tabulkách A.8 a A.9). Každý graf byl testován s hloubkou pokrytí 1 i 2, u algoritmů implementujících techniku PCT SmokeTest se STL „high“ i „medium“.

Vzhledem k tomu, že cílem implementace dalších algoritmů než jen existujícího PCT, bylo zahrnout do generování scénářů priority hran, obsahují všechny grafy nejméně jednu hranu s nejvyšší prioritou a nejméně jednu hranu se střední prioritou.

6.2.2 Hodnocení výsledku

PCT a PCT - PA

Cílem algoritmu PCT - PA bylo při podobném počtu scénářů a v nich obsažených hran, jako u scénářů generovaných algoritmem PCT, zvýšit obsazení hran s nejvyšší a střední prioritou.

Výsledný počet vygenerovaných scénářů byl v 58 % případech shodný, v 38 % případech byl schodný i počet hran ve scénářích. Kromě jednoho případu bylo vždy více vyprodukovaných scénářů u algoritmu PCT - PA, v nejhorším případě bylo tímto algoritmem vygenerováno o 4 scénáře víc. V rozdílu počtu hran není ani jeden algoritmus jednoznačně lepší, v nejhorším případě však byly algoritmem PCT nalezeny scénáře, jež měly dohromady o 14 hran méně, než hrany vygenerované ve scénářích algoritmem PCT - PA. V průměrném počtu hran s nejvyšší a střední prioritou ve scénářích byl úspěšnější algoritmus PCT - PA, který jich má 44,5 % , zatímco algoritmus PCT jich má pouze 41,7 %.

Porovnání scénářů vygenerovaných těmito dvěma algoritmy tedy potvrdilo, že algoritmus PCT - PA generuje scénáře s hranami s větší prioritou. Oproti algoritmu PCT je však testovacích scénářů více, a dohromady mají větší počet hran.

PCT SmokeTest a PCT SmokeTest II

Zatímco pro algoritmus PCT SmokeTest je určující délka scénáře, algoritmus PCT SmokeTest II při generování mimo délku zohledňuje i priority hran v grafu. PCT SmokeTest II by měl tedy pravděpodobně generovat scénáře s více hranami, ty však by měly mít větší prioritu.

Výsledek porovnání počtu vygenerovaných scénářů vykazuje, především v případě acyklických grafů, velmi podobné výsledky (celkem 94% shoda). V porovnání celkového počtu hran ve scénářích je úspěšnější (v 51,3 % případů) algoritmus PCT SmokeTest. Ve zbylých případech vrací oba algoritmy stejné výsledky. Naproti tomu PCT SmokeTest II generuje scénáře s větším průměrným počtem hran s nejvyšší nebo střední prioritou (54,1 % vůči 52,8 %).

Velký rozdíl je mezi algoritmy ve chvíli, kdy graf obsahuje cyklus, ve kterém je hrana s nejvyšší nebo střední prioritou. Ve scénářích vygenerovaných algoritmem PCT SmokeTest II je pak tento cyklus obsazen několikrát. Například u grafu 3 v tabulce A.1 byly pro hloubku pokrytí 1 nalezeny tyto scénáře:

1. `1 - 3 - 7 - 1 - 3 - 7 - 2 - 5 - 9 - 15 - 11 - 9 - 15 - 13 - 14 - 6 - 13 - 14 - 5 - 10 - 6 - 12`
2. `1 - 3 - 7 - 1 - 3 - 7 - 2 - 6 - 11 - 9 - 15 - 11 - 9 - 15 - 13 - 14 - 6 - 13 - 14 - 5 - 10 - 5 - 8`

Dohromady scénáře obsahují 45 hran, hned 19 z nich je tam obsaženo dvakrát a přitom pouze 6 z nich má prioritu nejvyšší, nebo střední. Oproti tomu PCT SmokeTest vygeneroval pro ten samý graf scénáře s pouhými 12 hranami, z nichž pouze dvě se opakují.

Porovnání scénářů vygenerovaných těmito dvěma algoritmy tedy ukázalo, že algoritmus PCT SmokeTest II sice, v souladu se specifikací, generuje scénáře s hranami s vyšší prioritou, avšak k tomu ve výsledku může přibýt tolik neprioritních hran že se v praxi tento algoritmus nemusí zcela uplatnit. Oproti tomu algoritmus PCT SmokeTest funguje naprosto spolehlivě a plně nahradí při vytváření scénářů člověka.

Kapitola 7

Závěr

V této bakalářské práci se mi povedlo vytvořit webovou aplikaci, která umožňuje uživateli v nahraném grafu vygenerovat různými algoritmy testovací scénáře. Testovací scénáře jsou důležitým prvkem pro správné testování funkčnosti software. Má aplikace se tedy uplatní v projektech, ve kterých je nezbytné vyvíjený software důkladně otestovat. Oproti existujícím aplikacím, které mají tuto funkci, nabízím při generování scénářů využití priorit akcí v testovaných průchodech aplikací, čímž se může testování lépe zacílit a v mnoha případech také zrychlit.

Algoritmy implementované v mé aplikaci se nazývají PCT, PCT - PA, PCT SmokeTest a PCT SmokeTest II. První dva realizují existující techniku pro generování testovacích scénářů Process Cycle Test, zbylé realizují dosud nedefinovanou techniku nazvanou PCT SmokeTest. Princip obou těchto technik, stejně jako zmíněných algoritmů, je v práci důkladně popsán. Z výsledků testování algoritmů vyplývá, že obzvlášť kombinace algoritmů PCT a PCT SmokeTest dává dohromady mocný nástroj pro testování aplikací, s možností zaměřit se buď na čas potřebný pro testování (což zajišťuje algoritmus PCT SmokeTest), nebo na komplexnost testu (to nabízí algoritmus PCT).

I pro mě samotného byla práce na této bakalářské práci velmi přínosná. Naučil jsem se, jak vytvořit JSP stránky, jak mohu z Javy komunikovat s databázovým systémem, jak správně pseudokódem dokumentovat fungování algoritmu a jak ve vytvářené aplikaci využít externích knihoven. V počátcích své práce jsem získal také mnohé zkušenosti v porozumění cizímu kódu, neboť jsem musel podrobně prostudovat desktopovou aplikaci PCTgen (vzniklou na Fakultě elektrotechnické ČVUT), do které se měly dát výsledné algoritmy snadno implementovat. Získal jsem také cenné informace o testování software. Nejdůležitější pro mne však bylo zjištění, že dokáži v krátké době, relativně bezchybně a bez cizí pomoci implementovat netriviální algoritmy.

Do budoucna je možné rozšířit mou práci o začlenění vyvinutých algoritmů do aplikace PCTgen, která by tím získala další velmi užitečnou funkci pro testování software.

Použitá literatura

[1] KOLÁŘ, Josef. *Teoretická informatika*. 2. vyd. Praha: Česká informatická společnost, 2000. ISBN 80-900-8538-5.

[2] TIM KOOMEN .. [ET AL.]. *TMap Next: for result-driven testing*. 's-Hertogenbosch: UTN Publishers, 2006. ISBN 90-721-9480-2.

[3] FOWLER, Martin. *UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition)*, 3. vyd. [s.l.] : Addison-Wesley Professional, 2003. ISBN 0-321-19368.

[4] BUREŠ, M. *Techniky pro návrh manuálních testů: průchody programem.* Přednáška předmětu TS1 na ČVUT, 2013.

[5] LÖWINGER, Lukáš. *Aplikace pro generování testovacích situací pro techniku Process Cycle Test.* Praha, 2014. Bakalářská práce. České vysoké učení technické v Praze.

[6] The Java Technology Phenomenon. In: *Oracle Java Documentation* [online]. Redwood City: Oracle Corporation and/or its affiliates (Oracle) [cit. 2016-04-23]. Dostupné z: http://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html

[7] The Java Programming Language Platforms. *Your First Cup* [online]. Redwood City: Oracle and/or its affiliates, 2012 [cit. 2016-04-24]. Dostupné z: http://docs.oracle.com/javaee/6/firstcup/doc/gkhoy.html

[8] TIOBE Index for April 2016. *Tiobe.com* [online]. TIOBE Software BV, 2016 [cit. 2016-04-19]. Dostupné z: http://www.tiobe.com/tiobe_index

[9] JavaServer Pages Technology. *The Java EE 5 tutorial: For Sun Java System Application Server 9.1* [online]. Redwood City: Oracle and/or its affiliates, 2010 [cit. 2016-04-19]. Dostupné z: http://docs.oracle.com/javaee/5/tutorial/doc/bnagx.html

[10] ZAMBON, Giulio. *Beginning JSP, JSF and Tomcat Java web development*. 2nd ed. New York, N.Y: Apress, 2012. ISBN 978-143-0246-244.

[11] About PostgreSQL. *PostgreSQL* [online]. The PostgreSQL Global Development Group [cit. 2016-04-23]. Dostupné z: http://www.postgresql.org/about/

[12] JDBC Overview. *Oracle Technology Network* [online]. Redwood City: Oracle Corporation and/or its affiliates (Oracle) [cit. 2016-04-23]. Dostupné z: http://www.oracle.com/technetwork/java/overview-141217.html

[13] Introduction. *JGraphX User Manual* [online]. JGraph Ltd [cit. 2016-04-23]. Dostupné z: https://jgraph.github.io/mxgraph/docs/manual_javavis.html

[14] Modules. *vis.js* [online]. Almende B.V. [cit. 2016-04-23]. Dostupné z http://visjs.org/index.html#modules

Příloha A

Výsledky testování

Vysvětlivky k tabulkám

Zkratky H, M, o a S u počtů hran značí jak jsou v nich zastoupeny jednotlivé priority. H („high“) značí počet hran s nejvyšší prioritou, M („medium“) se střední prioritu, o („other“) s nejmenší prioritou a S značí počet všech hran. Zkratka TDL je hloubka pokrytí („test depth level“) a zkratka STL značí „smoke test level“.

Název	Počet uzlů	Počet hran				Počet silně souvislých komponent
		H	M	O	S	
Graf 1	11	1	2	13	16	0
Graf 2	13	3	3	13	19	0
Graf 3	10	4	2	9	15	6
Graf 4	9	2	3	6	11	0

Tabulka A.1: *Informace o grafech, na kterých proběhlo testování správnosti generování*

#	Počet uzlů	Počet hran				Počet silně souvislých komponent
		H	M	O	S	
1	25	3	5	19	27	0
2	6	3	2	4	9	1
3	11	1	2	13	16	0
4	13	3	3	13	19	0
5	10	4	2	9	15	6
6	8	1	1	6	8	1
7	8	1	3	6	10	3
8	8	1	3	6	10	3
9	9	1	4	6	11	0
10	10	1	4	8	13	0
11	7	5	3	7	15	0
12	7	2	2	2	6	0
13	9	3	2	3	8	0
14	8	2	2	6	10	3
15	8	3	2	6	11	5
16	7	2	3	7	12	8
17	8	3	2	6	11	3
18	7	2	2	5	9	0
19	9	2	3	6	11	0
20	11	2	2	11	15	4

Tabulka A.2: *Informace o grafech, na kterých byla porovnávána efektivita algoritmů*

Výsledky ručního generování								
Název grafu	Graf 1				Graf 2			
Algoritmus	PCT		PCT - PA		PCT SmokeTest		PCT SmokeTest	
TDL	1	2	1	2	1	2	1	2
Počet scénářů	3	6	3	6	1	3	1	3
Počet hran	20	47	21	41	5	15	6	15
Název grafu	Graf 3							
Algoritmus	PCT		PCT - PA		PCT SmokeTest		PCT SmokeTest	
TDL	1	2	1	2	1	2	1	2
Počet scénářů	2	7	4	7	2	4	2	2
Počet hran	17	41	23	60	12	24	45	45
Název grafu	Graf 4							
Algoritmus	PCT		PCT - PA		PCT SmokeTest		PCT SmokeTest	
TDL	1	2	1	2	1	2	1	2
Počet scénářů	4	4	4	4	2	4	2	4
Počet hran	22	22	22	22	8	22	8	22

Tabulka A.3: *Výsledky ručního generování testovacích scénářů*

Výsledky automatického generování								
Název grafu	Graf 1				Graf 2			
Algoritmus	PCT		PCT - PA		PCT SmokeTest		PCT SmokeTest	
TDL	1	2	1	2	1	2	1	2
Počet scénářů	3	6	3	6	1	3	1	3
Počet hran	20	41	21	41	5	15	6	15
Název grafu	Graf 3							
Algoritmus	PCT		PCT - PA		PCT SmokeTest		PCT SmokeTest	
TDL	1	2	1	2	1	2	1	2
Počet scénářů	3	7	3	8	2	2	2	2
Počet hran	31	63	22	77	12	18	45	45
Název grafu	Graf 4							
Algoritmus	PCT		PCT - PA		PCT SmokeTest		PCT SmokeTest	
TDL	1	2	1	2	1	2	1	2
Počet scénářů	4	4	4	4	2	4	2	4
Počet hran	22	22	22	22	8	22	11	22

Tabulka A.4: *Výsledky automatického generování testovacích scénářů*

Algoritmus	TDL	STL	Počet scénářů																			
Číslo grafu			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
PCT	1		15	3	3	9	3	1	1	2	5	7	5	X	5	1	3	2	3	4	4	3
	2		16	3	6	13	7	1	1	3	7	17	8	6	5	1	4	2	3	3	4	6
PCT - PA	1		15	4	3	11	3	2	2	2	5	7	5	6	5	1	3	2	2	4	4	3
	2		17	6	6	17	8	2	2	3	7	17	10	6	5	1	6	3	3	4	5	6
PCT SmokeTest	1	H	2	1	1	1	2	1	1	1	1	1	1	2	3	1	1	1	1	1	2	1
	2	H	4	3	1	3	2	1	1	1	1	1	5	2	3	1	3	2	2	2	4	3
	1	M	4	2	2	3	2	1	3	2	2	2	2	4	4	1	1	1	1	2	2	1
	2	M	8	5	2	6	4	1	3	3	5	7	6	4	5	1	3	2	2	4	4	3
PCT SmokeTest II	1	H	2	2	1	1	2	1	1	1	1	1	1	2	3	1	1	1	1	1	2	1
	2	H	4	3	1	3	2	1	1	1	1	1	6	2	3	1	2	2	2	2	4	3
	1	M	4	2	2	3	2	1	3	2	2	2	2	4	4	1	1	1	1	3	2	1
	2	M	8	5	2	6	4	1	3	3	5	7	6	4	5	1	2	2	2	4	4	3

Tabulka A.5: *Počty scénářů vygenerované jednotlivými algoritmy pro testované grafy*

Algoritmus	TDL	STL	Počet hran ve scénářích																			
Číslo grafu			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
PCT	1		60	12	20	36	31	11	24	16	19	27	22	X	20	28	23	25	24	14	22	23
	2		67	19	41	61	63	11	24	28	28	36	34	6	20	20	61	38	26	12	22	69
PCT - PA	1		60	14	21	48	22	15	31	16	19	27	23	6	20	17	35	23	20	14	22	30
	2		69	31	41	75	77	15	31	28	28	36	43	6	20	17	46	40	26	14	22	52
PCT SmokeTest	1	H	9	6	6	5	12	4	4	4	3	3	5	2	12	9	7	4	6	4	8	7
	2	H	19	12	6	15	18	4	4	4	3	3	23	2	12	9	20	13	14	6	22	18
	1	M	16	7	11	15	14	11	26	16	7	7	10	4	16	12	9	9	10	6	11	12
	2	M	32	16	11	33	33	11	26	25	20	27	27	4	20	12	25	21	18	14	22	29
PCT SmokeTest II	1	H	10	17	6	6	45	11	11	7	3	3	5	2	12	12	16	16	14	4	11	14
	2	H	19	18	6	15	45	11	11	7	3	3	28	2	12	12	28	32	28	6	22	36
	1	M	16	17	11	17	44	11	26	16	7	7	10	4	16	12	16	16	14	10	11	14
	2	M	32	28	11	33	72	11	26	25	20	27	28	4	20	12	28	31	28	14	22	37

Tabulka A.6: *Počty hran ve scénářích, vygenerované jednotlivými algoritmy pro testované grafy*

Algoritmus	TDL	STL	Čas generování																			
Číslo grafu			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
PCT	1		0	0	0	0	0	0	1	0	0	0	1	0	0	1	0	1	1	1	1	0
	2		0	1	1	0	1	0	1	0	1	1	1	0	1	0	1	2	0	1	1	0
PCT - PA	1		1	1	1	1	0	1	6	0	0	0	1	0	0	1	0	1	1	1	1	0
	2		1	7	2	1	0	1	4	0	0	0	1	0	0	1	1	3	3	1	2	1
PCT SmokeTest	1	H	1	1	0	0	193	0	1	1	0	1	3	0	0	0	1	10	0	0	0	1
	2	H	5	1	0	1	137	0	0	0	0	0	17	0	0	0	5	56	0	0	1	1
	1	M	2	1	0	1	198	0	3	0	2	1	6	0	1	0	2	46	1	1	0	2
	2	M	3	4	1	2	513	0	1	2	1	6	19	0	0	0	7	799	1	0	1	5
PCT SmokeTest II	1	H	1	1	0	1	260	0	1	1	0	0	2	0	0	0	12	35	1	0	1	7
	2	H	5	1	0	3	129	0	0	0	0	0	9	0	0	0	13	49	2	0	2	10
	1	M	1	1	0	1	100	1	2	2	0	1	4	1	1	0	4	40	0	0	2	8
	2	M	3	2	0	3	383	0	3	7	1	3	15	1	0	0	12	192	1	1	2	8

Tabulka A.7: *Délka trvání generování scénářů jednotlivými algoritmy pro testované grafy (v milisekundách)*

Algoritmus	TDL	STL	Priority hran																															
Číslo grafu			1			2			3			4			5			6			7			8			9			10		
Priorita			H	M	O	H	M	O	H	M	O	H	M	O	H	M	O	H	M	O	H	M	O	H	M	O	H	M	O	H	M	O	H	M	O
PCT	1		17	8	35	6	1	5	1	3	16	5	7	24	7	3	21	1	1	9	1	5	18	1	3	12	3	9	7	3	13	11
	2		19	6	42	9	3	7	3	6	32	9	13	39	14	8	41	1	1	9	1	5	18	1	6	21	3	12	13	3	16	17
PCT - PA	1		17	8	35	7	2	5	2	3	16	11	9	28	5	4	13	2	1	12	2	6	23	1	3	12	3	9	7	3	13	11
	2		19	8	42	14	7	10	3	6	32	12	17	46	16	10	51	2	1	12	2	6	23	1	6	21	3	12	13	3	16	17
PCT SmokeTest	1	H	4	0	5	3	1	2	1	1	4	3	0	2	4	0	8	1	0	3	1	0	3	1	0	3	1	2	0	1	2	0
	2	H	8	1	10	6	1	5	1	1	4	6	3	6	6	0	12	1	0	3	1	0	3	1	0	3	1	2	0	1	2	0
	1	M	5	5	6	3	2	2	1	1	4	4	4	7	4	3	7	1	1	9	3	3	20	1	3	12	2	4	1	2	4	1
	2	M	10	7	15	8	3	5	1	3	7	6	8	19	8	6	19	1	1	9	3	3	20	1	5	19	2	9	9	2	13	12
PCT SmokeTest II	1	H	4	1	5	7	5	0	1	1	4	3	1	2	10	4	31	1	1	9	1	1	9	1	1	5	1	2	0	1	2	0
	2	H	8	1	10	8	3	7	1	1	4	6	3	6	10	4	31	1	1	9	1	1	9	1	1	5	1	2	0	1	2	0
	1	M	5	5	6	7	5	5	1	1	4	5	5	7	9	5	30	1	1	9	3	3	20	1	3	12	2	4	1	2	4	1
	2	M	10	7	15	12	7	9	1	3	7	10	8	15	15	9	48	1	1	9	3	3	20	1	5	19	2	9	9	2	13	12

Tabulka A.8: *Počty hran různých priorit ve scénářích, vygenerované jednotlivými algoritmy pro testované grafy (první část)*

Algoritmus	TDL	STL	Priority hran																												
Číslo grafu			11			12			13			14			15			16			17			18			19			20		
Priorita			H	M	O	H	M	O	H	M	O	H	M	O	H	M	O	H	M	O	H	M	O	H	M	O	H	M	O	H	M	O
PCT	1		6	4	12	X	X	X	3	6	11	4	5	19	9	3	11	5	7	13	9	3	12	4	3	7	4	7	11	4	3	16
	2		12	5	17	2	2	2	3	6	11	3	4	13	25	11	25	6	12	20	9	4	13	3	2	7	4	7	11	10	14	45
PCT - PA	1		12	4	7	2	2	2	3	6	11	3	3	11	13	18	14	5	6	12	7	3	10	4	3	7	4	7	11	5	4	21
	2		20	5	18	2	2	2	3	6	11	3	3	11	17	5	24	8	11	21	9	4	13	4	3	7	4	7	11	10	6	36
PCT SmokeTest	1	H	5	0	0	2	0	0	3	3	6	2	2	5	3	1	3	2	0	2	3	0	3	2	0	2	2	2	4	2	0	5
	2	H	13	4	6	2	0	0	3	3	6	2	2	5	8	1	11	5	2	6	7	0	7	3	1	2	4	7	11	5	0	12
	1	M	6	3	1	2	2	0	3	5	8	2	3	7	3	2	4	2	3	4	3	2	5	2	2	2	2	4	5	2	2	8
	2	M	13	5	9	2	2	0	3	6	11	2	3	7	9	4	12	5	6	10	7	2	9	4	3	7	4	7	11	5	4	20
PCT SmokeTest II	1	H	5	0	0	2	0	0	3	3	6	2	3	7	6	4	6	3	6	7	5	2	7	2	0	2	2	4	5	2	3	9
	2	H	20	3	5	2	0	0	3	3	6	2	3	7	10	7	11	6	12	14	9	5	14	3	1	2	4	7	11	5	8	23
	1	M	7	3	0	2	2	0	3	5	8	2	3	7	6	4	6	3	6	7	5	2	7	4	2	4	2	4	5	2	3	9
	2	M	18	4	6	2	2	0	3	6	11	2	3	7	10	7	11	6	11	14	9	5	14	4	3	7	4	7	11	5	8	24

Tabulka A.9: *Počty hran různých priorit ve scénářích, vygenerované jednotlivými algoritmy pro testované grafy (druhá část)*

Příloha B

Obsah přiloženého CD

Přiložené CD obsahuje následující složky a soubory:

- **javadoc -** složka s vygenerovanou dokumentací zdrojového kódu ve formátu HTML
- **projects -** složka s několika grafy vytvořenými aplikací PCTgen, které je možné nahrát do webové aplikace
- **src -** složka se zdrojovými soubory aplikace
- **tables -** složka se skripty pro vytvoření databázových tabulek
- **thesis -** složka se zdrojovými soubory pro LaTeX
- **web -** složka s JSP stránkami, kaskádovými styly, JavaScriptovými knihovnami a konfiguračními soubory
- **Automaticke_generovani_testovacich_scenaru_pro_procesy_webovych-_informacnich_systemu.pdf -** soubor s textem bakalářské práce
- **PCTwebapp.war -** archiv se spustitelnou verzí webové aplikace
- **PCTwebapp.xml -** konfigurační soubor pro nastavení připojení do databázového systému
- **Testovani_algoritmu.pdf -** soubor s výsledky testování algoritmů

Příloha C

Instrukce pro instalaci

C.1 Softwarové požadavky

Java

Na počítači s vhodným operačním systémem (Windows, Mac OS, Linux nebo Solaris) mít nainstalovanou aktuální verzi Java Runtime Enviroment (JRE), případně Java Development Kit (JDK).

Java EE Server

Na počítači mít nainstalovaný a správně nastavený Java EE Server, například Tomcat, GlassFish či WildFly.

PostgreSQL databáze

Pro činnost aplikace mít k dispozici databázový systém (například MySql či PostgreSQL) s odpovídajícím JDBC driverem pro Java EE Server.

C.2 Harwarové požadavky

Aplikace neklade na hardware žádné speciální požadavky a lze ji tedy spustit na jakémkoli běžně dostupném stroji.

C.3 Instalace a spuštění aplikace

1. Vytvořit novou databázi pro účely této aplikace.
2. Podle skriptů `Algorithm.txt`, `Graph.txt` a `Test_scenario.txt` ve složce `tables` na přiloženém CD vytvořit v databázi příslušné databázové tabulky.
3. Naplnit daty databázovou tabulku `Algorithm` pomocí příkazů v souboru `Algorithm_Values.txt` ve složce `tables`.

4. Vyplnit v souboru `PCTwebapp.xml` přihlašovací údaje do databázového systému (změnit hodnoty ohraničené hranatými závorkami a tyto závorky smazat).

5. Umístit do vhodné složky na serveru archiv `PCTwebapp.war` (u serveru Tomcat je to složka `webapps`).

6. Umístit do vhodné složky na serveru konfigurační soubor `PCTwebapp.xml` (u serveru Tomcat je cesta k této složce: `$CATALINA_HOME/conf/[engine]/[domena]/`).

7. Spustit instanci Java EE Serveru (u serveru Tomcat běžícího v OS Windows spuštěním souboru `bin\startup.bat`, v OS Linux nebo Mac OS `bin/startup.sh`).

Pokud vše proběhlo správně, je možné na adrese `/nazev_aplikace/index.jsp` navštívit titulní stránku aplikace. Na té si uživatel vybere soubor s grafem, který chce nahrát, a zmáčkne tlačítko „upload graph“. Nahraný graf se následně objeví v tabulce na titulní stránce aplikace. Vygenerovat pro graf testovací scénáře je poté možné kliknutím na tlačítko „view“ u položky s daným grafem a postupem podle informací uvedených na nově zobrazené stránce.