FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Aplikace pro demonstraci funkce simulovaného ochlazování
Student:	Bc. Michal Kluzáček
Vedoucí:	doc. Ing. Petr Fišer, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

Vytvořte aplikaci pro demonstraci funkce simulovaného ochlazování (SA). Aplikace bude použita pro výuku předmětu MI-PAA.
Základní pedagogické požadavky jsou:
- Přehledné a snadno použitelné grafické rozhraní, aplikace musí být jednoduše spustitelná, bez nutnosti jakékoliv instalace.
- Možnost výběru z několika problémů k řešení, automatické i ruční generování instancí problémů, možnost práce s více instancemi, vracení se k nim.
- Možnost nastavování parametrů SA, výběru ochlazovacího rozvrhu, implementace adaptačních mechanizmů.
- Sledování běhu algoritmu (graficky), vytváření statistik.

Nejprve proveďte důkladnou analýzu funkčních i ostatních (nefunkčních) požadavků. Za tímto účelem proveďte uživatelský průzkum mezi absolventy předmětu MI-PAA a jeho cvičícími.

Dále navrhněte vhodnou SW architekturu pro implementaci. Zde mějte na zřeteli univerzálnost a rozšiřitelnost (pro další problémy).
Nakonec nástroj implementujte, adekvátně otestujte, proveďte důkladné uživatelské testy.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 24. prosince 2017

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Aplikace pro demonstraci funkce simulovaného ochlazování

Bc. Michal Kluzáček

Katedra softwarového inženýrství
Vedoucí práce: doc. Ing. Petr Fišer, Ph.D.

6. května 2018

Poděkování

V první řadě bych rád poděkoval doc. Ing. Petru Fišerovi, Ph.D. za vedení této práce. Dále bych chtěl poděkovat Jaroslavu Veselému a Adamu Kuglerovi za spolupráci na vyvíjené aplikaci, bez jejich přispění by aplikace nevypadala tak, jak vypadá. V poslední řadě bych rád poděkoval všem, kteří si našli čas a pomohli nám s otestováním výsledné aplikace.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 6. května 2018 .....

Odkaz na tuto práci

Kluzáček, Michal. *Aplikace pro demonstraci funkce simulovaného ochlazování.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

V práci se zabývám tvorbou výukové webové aplikace pro demonstraci běhu simulovaného ochlazování. Přesněji tedy analýzou, návrhem, implementací a v poslední řadě také testováním. Implementoval jsem několik druhů problémů, jako jsou například problém obchodního cestujícího nebo problém minimálního uzlového pokrytí. Dále jsem ke každému z problémů vytvořil generátor instancí, schopný generovat instance daného problému na základě zvolených parametrů.

Klíčová slova simulované ochlazování, iterativní algoritmy, webová aplikace

Abstract

This thesis is about creating an educational web application for demonstration of the simulated annealing algorithm. More precisly about analysis, design, implementation and testing of this application. I implemented multiple problems like travelling salesman problem and minimum vertex cover problem. I also created generators for each problem, that can generate instances of the problem based on input parameters.

Keywords simulated annealing, iterative algorithms, web application

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V rámci předmětu MI-PAA se studenti Fakulty informačních technologií učí základy iterativních algoritmů. Tyto algoritmy však nepatří mezi jednoduché a jejich fungování může být ze začátku matoucí. Proto je dobré mít pomůcku, program, na kterém si student může vyzkoušet, jak tyto algoritmy fungují a jak jednotlivé parametry ovlivňují jejich průběh.

Bývalí studenti proto takovéto programy, v rámci svých závěrečných prací, vytvořili. Bohužel technologie využité při jejich implementaci, kterými byly javovské applety, jsou dnes již zastaralé a na novějších internetových prohlížečích je nelze spustit. Za další problém považuji, že jednotlivé metody jsou od sebe oddělené a implementované v rámci samostatných aplikací, tudíž si student nemůže vyzkoušet všechny probírané metody na jednom místě.

Z těchto důvodů bylo třeba vytvořit něco nového, a protože mě tento předmět bavil, rozhodl jsem se, že se na tomto vývoji chci podílet. Vzhledem k tomu, že bylo třeba implementovat ne jednu, ale tři metody, byla tato práce rozdělena mezi další dva studenty, jmenovitě: Jaroslava Veselého a Adama Kuglera. Já se ve své práci zaměřuji na metodu simulovaného ochlazování, což byla metoda, kterou jsem implementoval v rámci předmětu MI-PAA, a moji kolegové se zaměřují na tabu prohledávání [1] a genetický algoritmus [2].

Aby byly všechny tyto metody pohromadě a řešení bylo celistvé, rozhodli jsme se vytvořit jednu webovou aplikaci, která bude obsahovat všechny tyto metody a konzistentní uživatelské rozhraní napříč celou aplikací.

Kapitola 1

Cíl práce

Cílem práce je vytvořit aplikaci pro demonstraci běhu algoritmu simulovaného ochlazování, která bude použita v rámci výuky předmětu MI-PAA.

Na začátku práce je nutné provést sběr požadavků, což provedu za pomoci dotazníku cíleného na absolventy a cvičící předmětu MI-PAA. Poté provedu průzkum již existujících aplikací, což mi poskytne další informace pro navržení systému. Všechny získané informace budou základem pro sepsání požadavků na systém, na jejichž základě budou vybrány technologie na jeho implementaci.

Dále provedu návrh systému, a to jak architektury, tak uživatelského rozhraní.

Poté systém implementuji a následně otestuji.

V rámci některých těchto částí budu spolupracovat s kolegy, kteří implementují zbylé metody.

Základní požadavky na systém:

- Rozšiřitelnost
  - k systému musí být možné snadno přidávat další problémy, případně metody.
- Jednoduchost
  - systém musí být jednoduchý na ovládání a bez nutnosti jakékoliv instalace.
- Volba parametrů
  - systém musí umožňovat nastavení všech základních parametrů simulovaného ochlazování.
- Sledování běhu (graficky)
  - systém musí graficky zobrazovat průběh řešení.

- Generování instancí
  - systém musí umožnit generovat instance všech implementovaných problémů.
- Možnost práce s více instancemi
  - systém musí umožňovat práci s více instancemi a možnost se k nim vracet.

Kapitola 2

Analýza a návrh

Tato kapitola bude o analýze a návrhu systému. Nejdříve zhodnotím výsledky dotazníku, získané od absolventů předmětu MI-PAA. Poté provedu průzkum podobných již existujících aplikací. Na základě těchto poznatků popíši požadavky na systém, a to jak funkční, tak nefunkční. Poté bude následovat popis použitých technologií, struktury navrženého systému a na závěr návrh uživatelského rozhraní.

2.1 Uživatelský průzkum

Po sepsání základních požadavků jsme vytvořili dotazník cílený na absolventy předmětu MI-PAA, abychom zjistili, co se jim líbilo/nelíbilo na původních aplikacích, ale také, co by rádi viděli v naší nové aplikaci.

Dotazník jsme zaslali všem absolventům z našeho ročníku a získali tak 29 odpovědí.

Podíváme se tedy na výsledky položených otázek a na to, jaký dopad měly na návrh řešení.

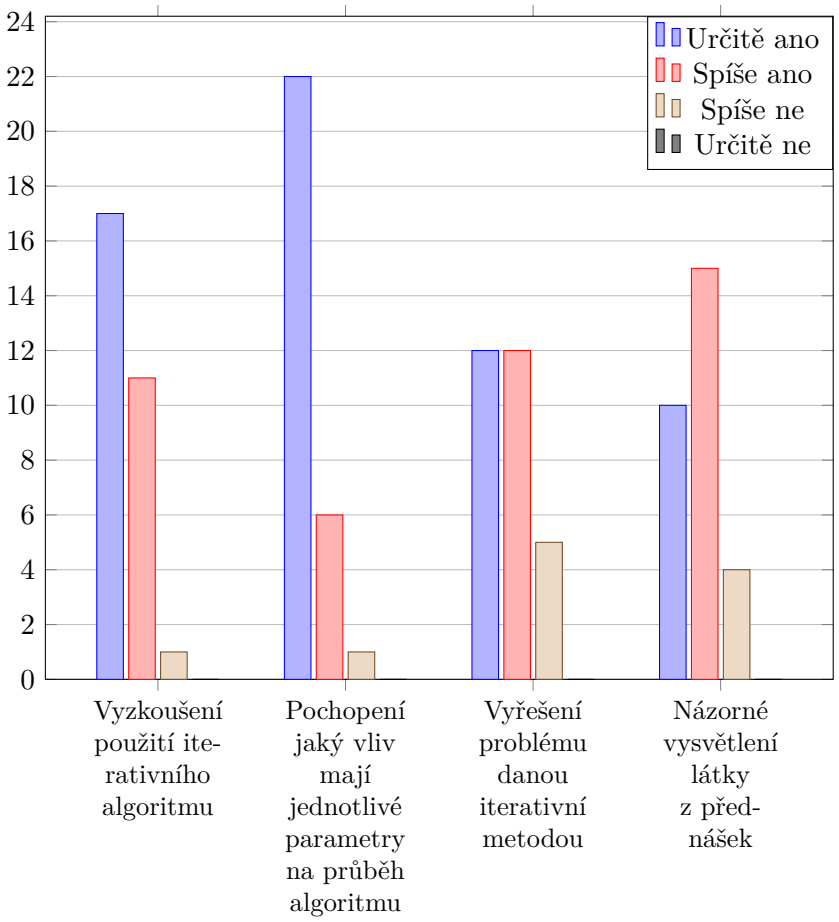
2.1.1 Položené otázky

Zde jsou otázky, na které jsme se studentů zeptali.

Jak moc (by) přispěly následující možnosti k pochopení iterativních metod?

Tato otázka byla zaměřena na zjištění toho, co nejvíce přispělo k pochopení probíraných metod a zda názorná ukázka běhu za pomoci aplikace má smysl.

Odpovědi:

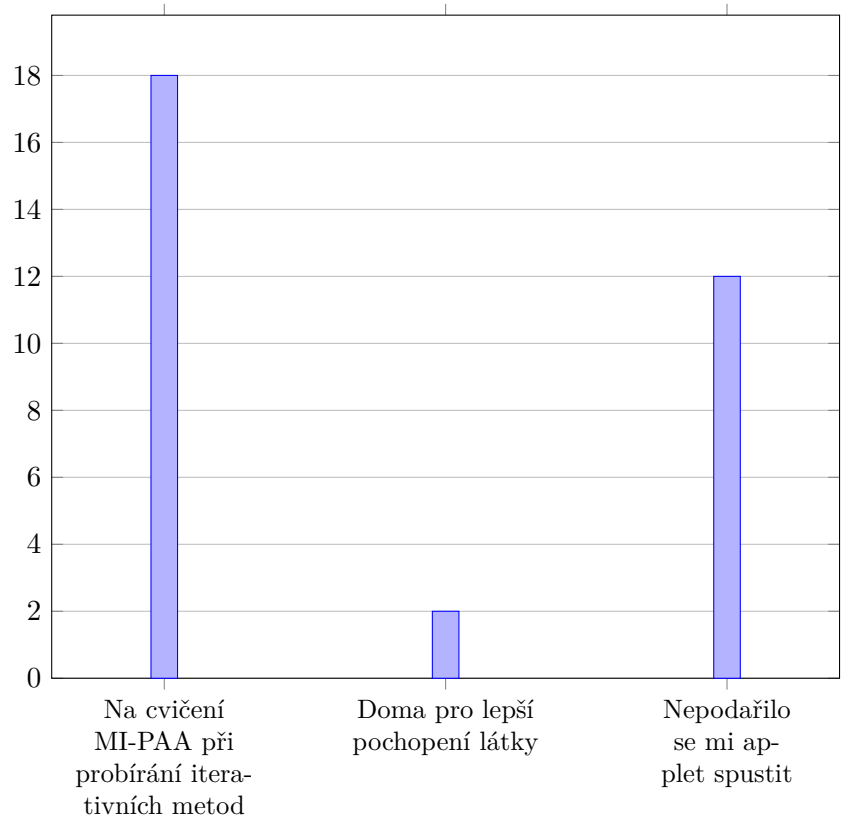


Závěr: Z výsledků vidíme, že vyzkoušení iterativního algoritmu, možnost nastavení jednotlivých parametrů a sledování jejich vlivu na průběh algoritmu u většiny studentů výrazně přispělo k pochopení vyučovaných metod.

Při jaké příležitosti jste původní Java applety použil(a)?

Účelem této otázky bylo zjistit, zda studenti aplikaci použili i při jiné příležitosti než na cvičení, kde se probírala daná metoda.

Odpovědi:

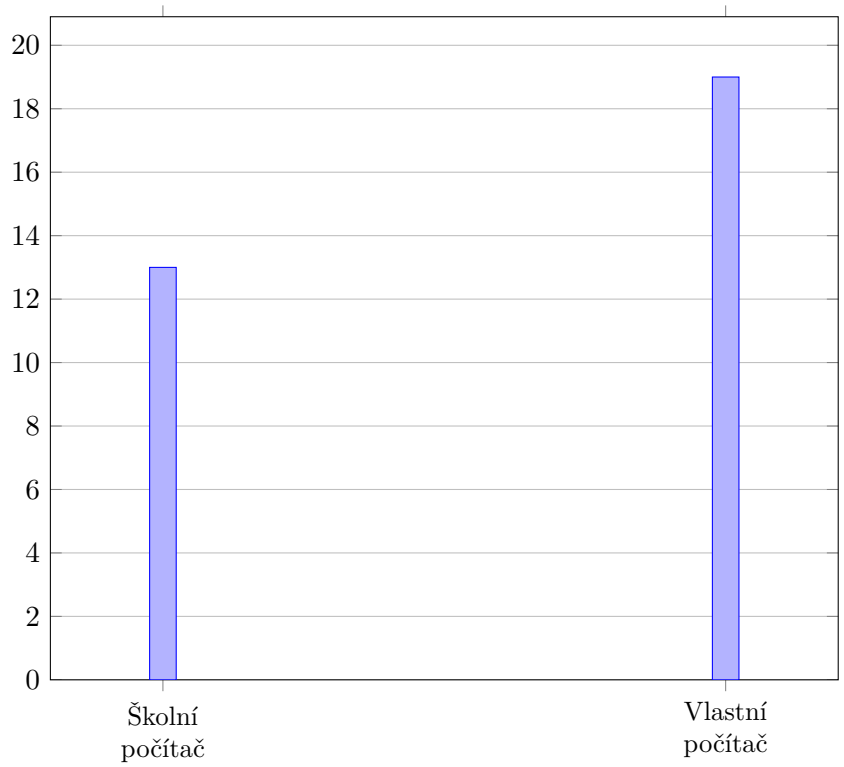


Závěr: Většina studentů použila applety především na cvičení, kde byly probírány iterativní metody, což je nejspíše způsobeno tím, že applety nejsou spustitelné na nových prohlížečích, které mají studenti na svých počítačích. Tomu nasvědčuje i to, že téměř 50 % dotázaných se nepodařilo applet spustit.

Na jakém zařízení jste aplikaci použil(a) nebo kde jste se ji pokusil(a) spustit?

Cílem této otázky bylo zjistit, kde studenti původní aplikace používali, což může mít vliv na volbu platformy. Pokud by je například studenti pouštěli pouze na školních počítačích, je možnost vytvořit desktopovou aplikaci mnohem přívětivější. Otázka umožňovala volbu více odpovědí najednou v případě, že student zkoušel více možností.

Odpovědi:

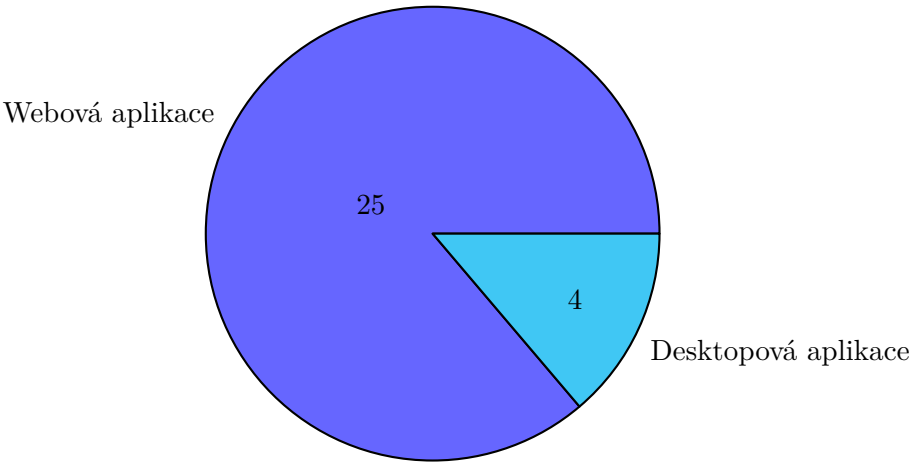


Závěr: Studenti používali původní aplikace jak na svých, tak na školních počítačích, proto je lepší zvolit platformu, která je snadno dostupná ze všech počítačů.

Dáváte přednost webové nebo desktopové aplikaci?

Velice důležitá otázka pro správnou volbu platformy. Obě možnosti mají své klady a zápory, proto bylo nutné zjistit, čemu by studenti dali přednost.

Odpovědi:

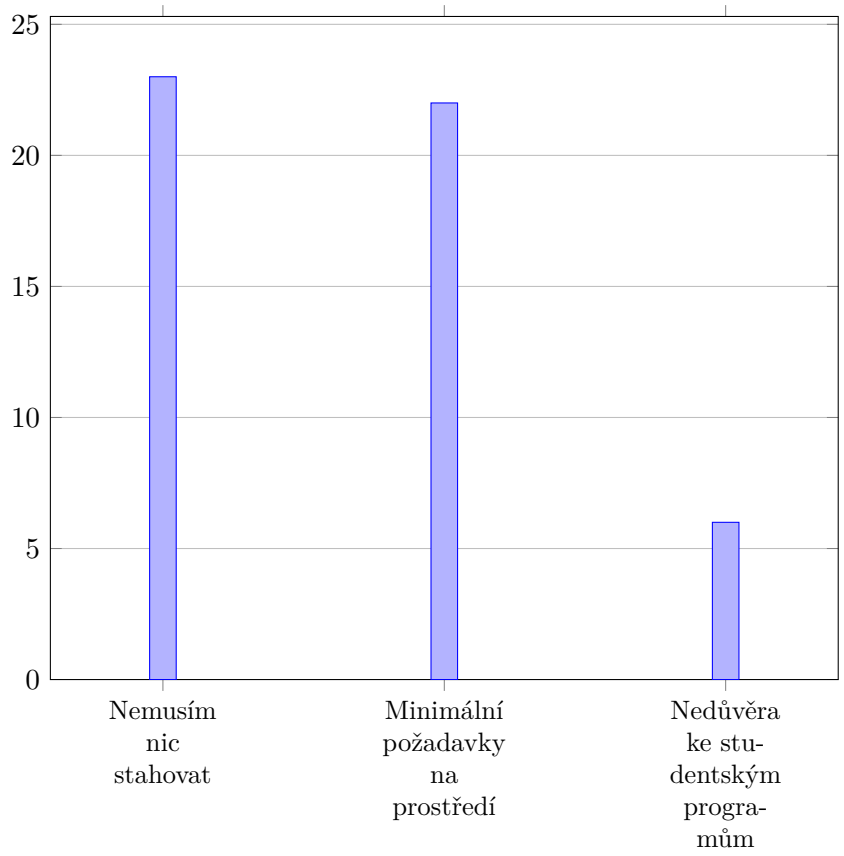


Závěr: Studenti dávají značnou převahou přednost webové aplikaci.

Proč dáváte přednost webové aplikaci?

Na základě předchozí odpovědi bylo také potřeba zjistit, z jakého důvodu dal student přednost webové aplikaci. Jinými slovy, co považuje za hlavní výhody této možnosti.

Odpovědi:

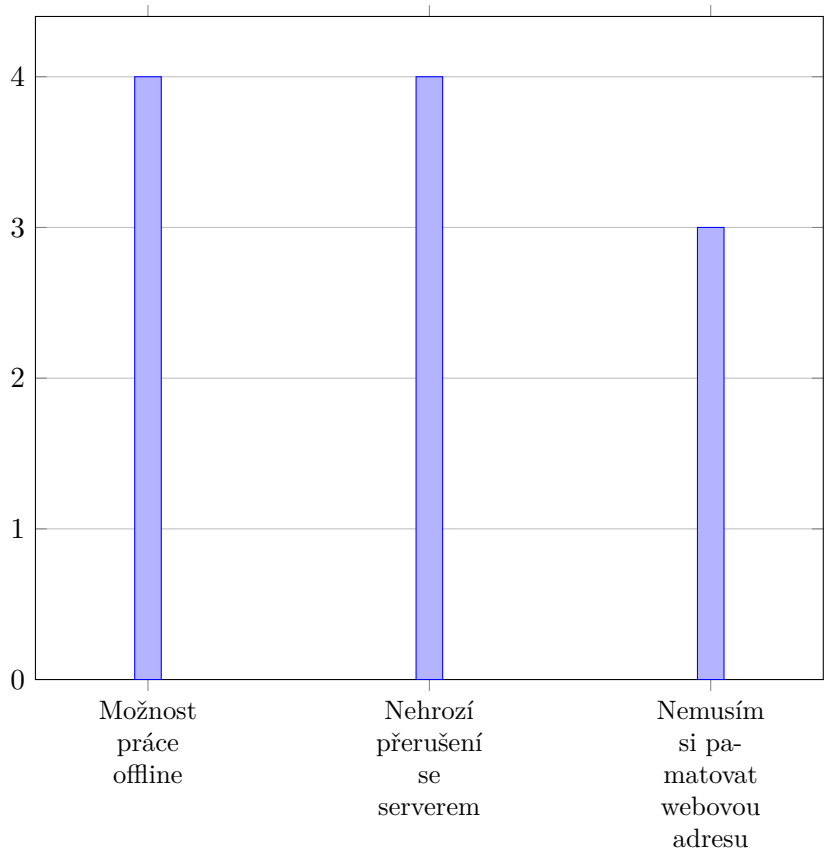


Závěr: Za hlavní výhody studenti považují jednoduchost použití webové aplikace, která vychází z absence stahování či instalace, ať už samotné aplikace, nebo jiného softwaru nutného k jejímu spuštění.

Proč dáváte přednost desktopové aplikaci?

Stejná otázka jako předchozí, ale pro studenty, kteří by dali přednost desktopové aplikaci.

Odpovědi:



Závěr: Hlavní výhodou této volby je možnost práce bez internetového připojení, s čímž souvisí i bezpečí ve smyslu ztráty práce při přerušení internetového spojení.

2.1.2 Shrnutí získaných informací

V této části shrnu získané informace a jejich dopad na návrh systému.

1. Co přispělo k pochopení probíraných metod?

   Zjištění: Kromě samotného výkladu a vyzkoušení implementace vybrané metody uvedla většina studentů, že velký vliv na pochopení probíraných metod měla právě možnost vyzkoušet si jednotlivé algoritmy. A to především díky vizualizaci, kde mohli sledovat vývoj řešení a vliv jednotlivých parametrů na celý průběh algoritmu.

   Důsledek: Na základě tohoto zjištění je třeba mít v aplikaci přehlednou grafickou vizualizaci běhu algoritmu. Dále je nutné, aby si student mohl u každé metody volit všechny základní parametry a sledovat jejich

vliv na celý průběh algoritmu. Pro lepší porovnání jednotlivých rozdílů bude možné výsledky jednotlivých běhů prokládat, což proces porovnání velmi usnadní.

2. Jaké zařízení by měla aplikace podporovat?

   Zjištění: U této otázky jsme zjistili, že studenti spouštěli původní aplikace jak na svých, tak na školních počítačích. Velkému množství studentů se nepodařilo aplikace ani spustit, a to kvůli zastaralé technologii java appletů.

   Důsledek: Vytvoření desktopové aplikace, která by byla nainstalována na školních počítačích, nedává smysl. Je třeba vytvořit aplikaci, která bude snadno spustitelná na všech počítačích. Měla by být použita technologie, u které se předpokládá, že bude podporována i v následujících letech.

3. Webová, nebo desktopová aplikace?

   Zjištění: Studenti jednoznačně dávají přednost webové aplikaci. Hlavním důvodem pro toto rozhodnutí je jednoduchost a nenáročnost na rozhraní.

   Důsledek: Zvolenou platformou pro náš systém bude webová aplikace, která bude dostupná bez nutnosti přihlášení či instalace.

V závěru by se tedy mělo jednat o webovou aplikaci, která bude využívat technologie podporované i v nadcházejících letech. Tato aplikace musí obsahovat grafické rozhraní zobrazující průběh aktuálně řešeného problému, aby mohl student sledovat jeho vývoj a změny způsobené rozdílným nastavením parametrů.

2.2 Rešerše konkurenčních systémů

Vzhledem k specifické povaze našeho projektu není příliš aplikací, které by se daly považovat za přímou konkurenci. Proto se zde zaměřím na aplikace graficky znázorňující průběh algoritmů a na původní applety, které se momentálně při výuce používají a jsou nejblíže funkcionalitě našeho systému.

U každého systému obecně popíši jeho funkcionalitu a poté rozeberu jeho klady a zápory. Na závěr shrnu všechny zmíněné systémy.

2.2.1 Applet simulovaného ochlazování

Původní aplikace navržená pro demonstraci běhu simulovaného ochlazování. Jedná se o systém v současnosti používaný při výuce předmětu MI-PAA. Systém používá zastaralou technologii java appletů, která není dnešními prohlížeči podporována. Applet je dostupný na stránce `http://ddd.fit.cvut.cz/PAA/SA/applet_window.html`. Screenshot appletu viz obrázek 2.1.

2.2.1.1 Klady

Mezi klady tohoto systému patří:

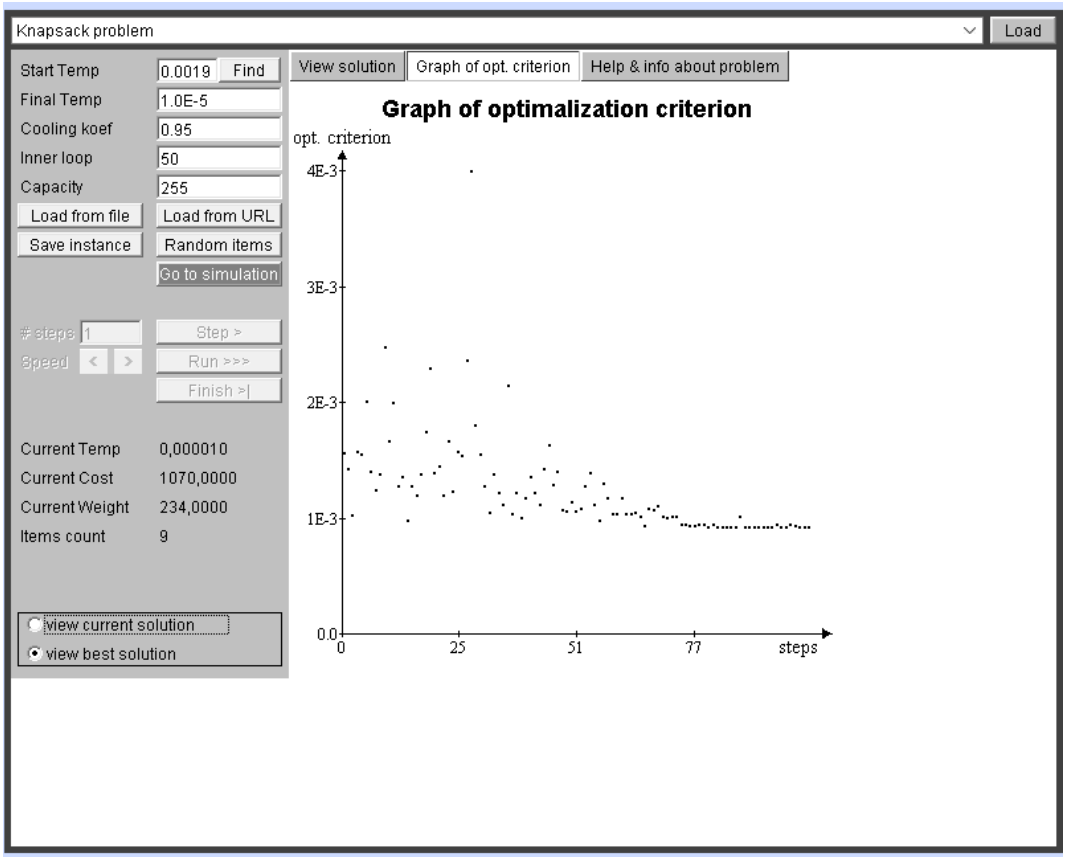
- vizualizace algoritmu,
- podpora řešení vícero druhů problémů,
- nastavování parametrů,
- generátor instancí,
- možnost nahrání vlastní instance,
- výpočet počáteční teploty,
- kontrola vstupu.

2.2.1.2 Zápory

Mezi zápory patří:

- zastaralá technologie appletů,
- jednotlivé body v grafu po najetí myší nezobrazí podrobné údaje konfigurace (například hodnotu optimalizačního kritéria),
- nastavení rychlosti výpisu nikde neukazuje aktuální hodnotu tohoto atributu,
- není responzivní,
- chybí popisky u jednotlivých parametrů,
- tlačítko „Go to simulation“, na které musíte kliknout před tlačítkem „Run“.

Hlavní nevýhodou je zde určitě technologie appletů, kterou je na dnešních prohlížečích téměř nemožné spustit. Jediná možnost, pomocí jíž se mi podařilo aplikaci otevřít, byla stará verze prohlížeče Mozilla Firefox, která se však bohužel aktualizovala. Když jsem se snažil aplikaci spustit nyní, neuspěl jsem ani po několika hodinách snažení a nemalém množství návodů, které jsem na internetu našel. Dalším problémem je celkově zastaralý vzhled aplikace, který dnes na webu normálně nenajdete.



Obrázek 2.1: Simulované ochlazování, původní applet

2.2.2 Applet genetického algoritmu

Druhý z appletů, který však jako metodu řešení problémů využívá genetický algoritmus. Velice se podobá předchozímu appletu jak funkcionalitou, tak vzhledem. Applet je dostupný na stránce `http://ddd.fit.cvut.cz/PAA/GA/`. Screenshot appletu viz obrázek 2.2.

2.2.2.1 Klady

Mezi klady tohoto systému patří:

- vizualizace algoritmu,
- podpora řešení vícero druhů problémů,
- nastavování parametrů,
- pokročilé mechanizmy,

- možnost upravit vstupní instance přímo v aplikaci,
- možnost nahrání vlastní instance,
- kontrola vstupu.

2.2.2.2 Zápory

Mezi zápory patří:

- zastaralá technologie appletů,
- jednotlivé body v grafu po najetí myší nezobrazí podrobné údaje konfigurace (například fitness všech jedinců: nejlepší, nejhorší, průměrný),
- nevaruje uživatele již při zadání nesprávných hodnot atributů, ale až po spuštění, rovněž jsou varováni v nečitelné formě („java.lang.exception“),
- není responzivní,
- vizualizace obchodního cestujícího je nečitelná,
- chybí popisky u jednotlivých parametrů,
- dropboxy „Configuration“ a „Help“ jsou matoucí (neznámá funkčnost), funkce u „Help“ vůbec nefungují.

2.2.3 Travelling salesman with simulated annealing

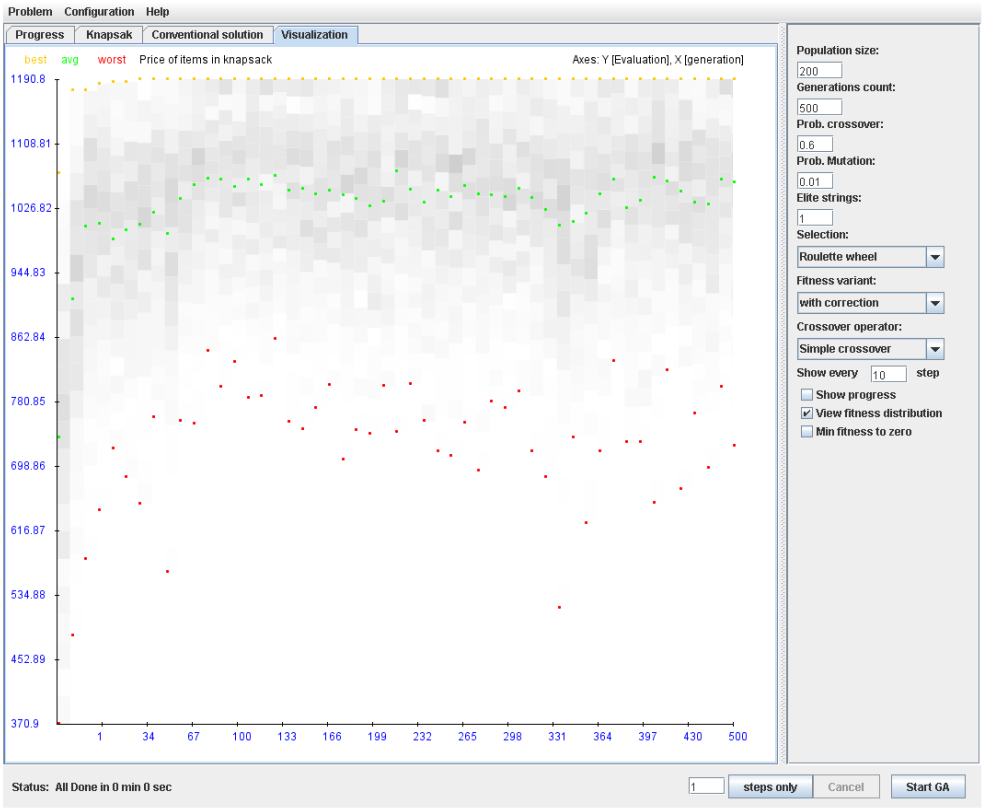
Jedná se o stránku, která kromě úvodu do problému obchodního cestujícího a popisu metody simulovaného ochlazování obsahuje demo, ve kterém je možné si vyzkoušet řešení tohoto problému za pomoci simulovaného ochlazování.

Toto řešení pak velmi pěkně zobrazuje přímo na mapě, kde můžeme sledovat vývoj cesty. Kromě mapy také ukazuje graf vývoje řešení a umožňuje nastavení řady parametrů včetně výběru samotných měst. Aplikace je dostupná na stránce `http://toddwschneider.com/posts/traveling-salesman-with-simulated-annealing-r-and-shiny/`. Obrázek aplikace viz 2.3.

2.2.3.1 Klady

Mezi klady tohoto systému patří:

- velice pěkná vizualizace algoritmu,
- nastavení parametrů,
- možnost měnit instanci problému (vybírat města).



Obrázek 2.2: Genetický algoritmus, původní applet

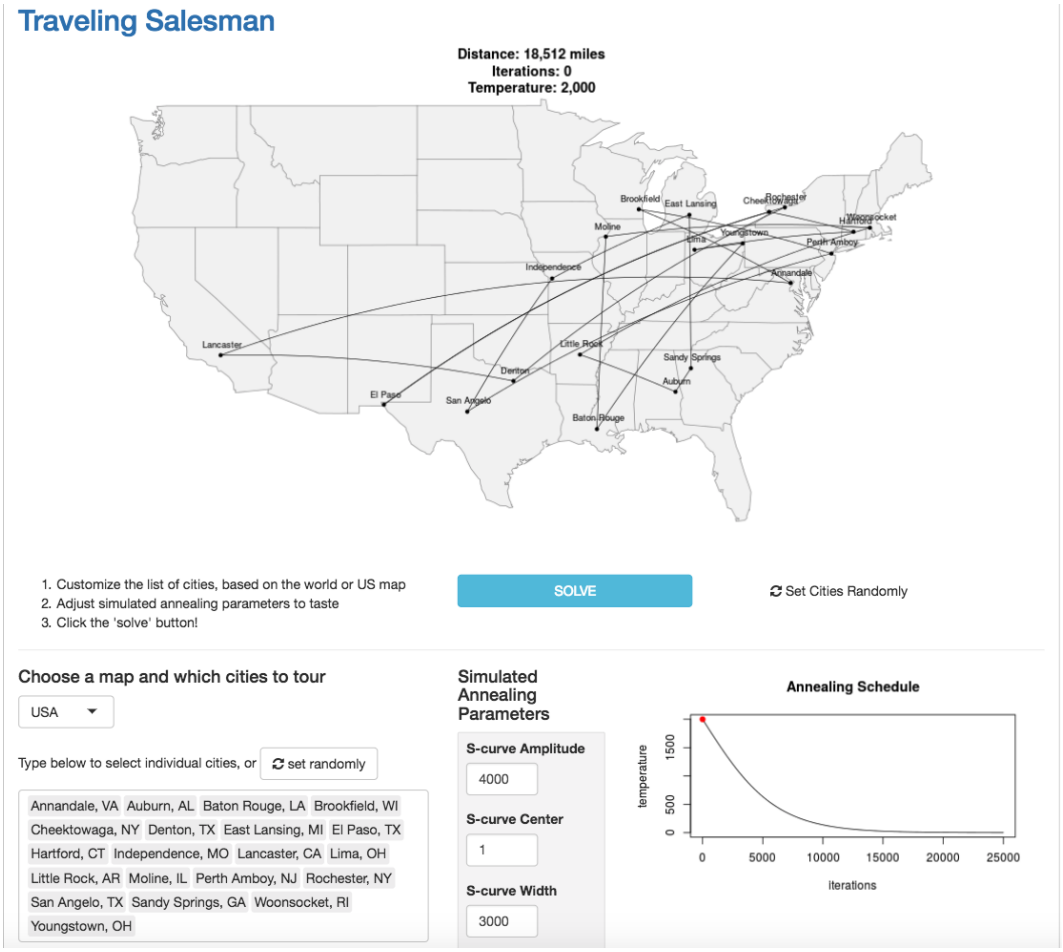
2.2.3.2 Zápory

Mezi zápory patří:

- chybí popisky u jednotlivých parametrů,
- nepříliš vhodné rozložení komponent,
- běh není možné předčasně ukončit,
- řeší pouze jediný problém.

2.2.4 Vjeux simulated anneling

V tomto případě se jedná o studentský projekt, jehož cílem bylo využít simulované ochlazování na řešení problému. Tímto problémem je mřížka, která je však „zamotána“ a je třeba ji rozmotat. Celá aplikace je velmi pěkně



Obrázek 2.3: Travelling salesman with simulated annealing

graficky řešena, celé rozhraní je velice čisté a dobře čitelné. Aplikace zobrazuje jak mřížku, která se postupně upravuje, tak vývoj ceny. Během výpočtu je také možné sledovat statistiky, jako jsou: cena, počet vyzkoušených stavů, počet přijatých stavů a čas. Aplikace je dostupná na stránce `http://fooo.fr/~vjeux/epita/recuit/recuit.html#` a lze ji vidět na obrázku 2.4.

2.2.4.1 Klady

Mezi klady tohoto systému patří:

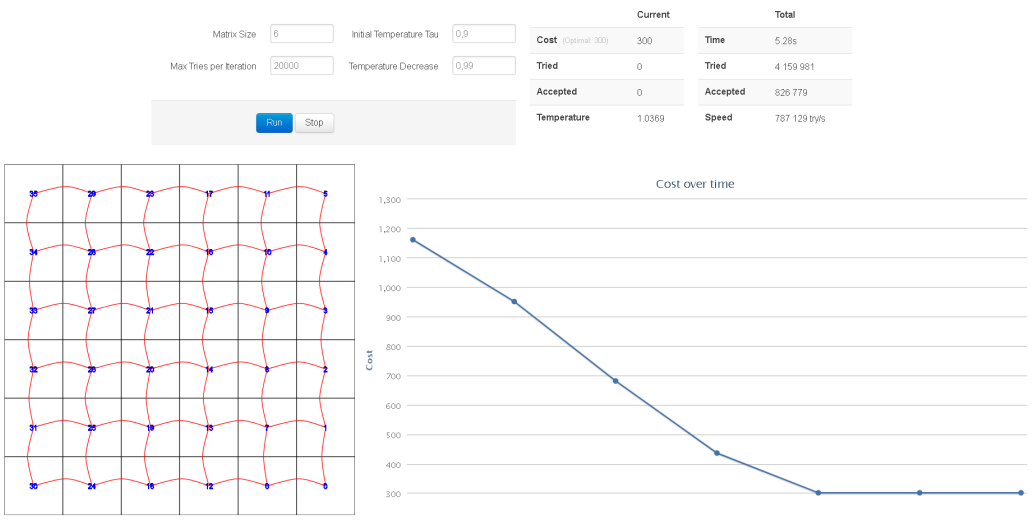
- velice pěkná vizualizace algoritmu,
- nastavení parametrů,

- výpis statistik,
- velmi čisté uživatelské rozhraní,
- možnost měnit instanci problému.

2.2.4.2 Zápory

Mezi zápory patří:

- chybí popisky u jednotlivých parametrů,
- při malém vnitřním cyklu se statistiky mění příliš rychle a jsou nečitelné,
- nevaruje při zadání neplatných parametrů (záporné hodnoty),
- řeší pouze jediný problém.



Obrázek 2.4: Vjeux simulated anneling

2.2.5 Shrnutí

Jak můžeme vidět v tabulce 2.1, původně používané systémy se liší od ostatních aplikací dostupných na webu. Hlavním rozdílem těchto dvou skupin je, že aplikace dostupné na webu se zaměřují na řešení jednoho typu problému, nikoliv na vícero. Každý systém také podporuje formu grafické vizualizace průběhu výpočtu a možnost řešení různých instancí. Aplikace nalezené na webu jsou velmi snadno spustitelné bez jakéhokoliv instalování nebo nastavování. V čem mají všechny aplikace problém je to, že neobsahují žádné vysvětlivky / popisy

Tabulka 2.1: Shrnutí systémů

	Applet SA	Applet GA	Salesman	Vjeux
Vícero problémů	✓	✓	✗	✗
Vizualizace	✓	✓	✓	✓
Nastavení parametrů	✓	✓	✓	✓
Vysvětlivky	✗	✗	✗	✗
Generování instancí	✓	✓	✓	✓
Snadno spustitelné	✗	✗	✓	✓

jednotlivých parametrů, takže člověk, který se v dané problematice nepohybuje, neví, co vlastně nastavuje.

Naše aplikace se tedy bude lišit od konkurence tímto:

- řešení vícero druhů problémů,
- snadná spustitelnost,
- vysvětlivky u jednotlivých parametrů a problémů,
- historie a porovnání dřívějších výpočtů,
- 3 metody na řešení problémů.

2.3 Požadavky na systém

V této sekci podrobněji vysvětlím, jaké požadavky má systém splňovat.

2.3.1 Funkční požadavky

Funkční požadavky jsou, jak už název napovídá, požadavky na funkcionalitu systému, tedy na to, co by měl daný systém umět.

2.3.1.1 Volba parametrů

Systém musí umožňovat nastavení všech parametrů metody simulovaného ochlazovaní, tedy:

- počáteční teplota,
- ochlazovací koeficient,
- velikost vnitřního cyklu,
- konečná teplota.

2.3.1.2 Adaptační mechanizmy

Metoda simulovaného ochlazování musí obsahovat adaptační mechanizmy, především výpočet počáteční teploty.

2.3.1.3 Sledování průběhu

Systém musí umožnit sledovat průběh algoritmu v grafické podobě (vývoj řešení).

2.3.1.4 Statistky

Systém musí být schopen zobrazit statistiky běhu, tedy například:

- počet iterací,
- nalezené řešení,
- doba běhu,
- zvolené parametry spuštěného řešení.

2.3.1.5 Možnost práce s více instancemi

Systém musí umožňovat práci s více instancemi. K těmto instancím musí být možné se vrátit.

2.3.1.6 Vícero problémů

Systém musí být schopen pracovat s více problémy, ne ve smyslu většího množství instancí, ale s větším počtem typů problémů. Mezi problémy patří například:

- problém batohu,
- SAT problém,
- problém obchodního cestujícího,
- problém minimálního pokrytí.

2.3.1.7 Generování instancí

Systém musí umožňovat generování instance problému, a to jak ručně, tak automaticky. Přičemž ručně znamená například pomocí textového souboru, ve kterém uživatel pomocí specifického formátu nadefinuje danou instanci. Automatické generování takovýto soubor vytvoří na základě zvolených parametrů.

2.3.1.8 Historie

Systém bude uchovávat historii dříve řešených problémů, u kterých bude možné znovu zobrazit výsledky a graf průběhu.

Tyto výsledky bude možné připojit k jiné vyřešené instanci, aby bylo možné je porovnat.

2.3.2 Nefunkční požadavky

Nefunkční požadavky jsou takové požadavky, které nekladou nároky na funkcionalitu sytému, ale například na platformu, dostupnost nebo výkon.

2.3.2.1 Podpora prohlížečů

Systém bude kompatibilní s těmito internetovými prohlížeči:

- Chrome,
- Mozilla Firefox,
- Microsoft Edge,
- Opera.

U těchto prohlížečů budou podporovány dvě poslední stabilní verze.

2.3.2.2 Webová služba

Systém bude dostupný přes webové prohlížeče.

2.3.2.3 Rozšiřitelnost

Systém musí být snadno rozšiřitelný o nové typy problémů.

2.3.2.4 Instalace

Systém musí být spustitelný bez nutnosti jakékoliv instalace, s výjimkou dříve zmíněného webového prohlížeče.

2.4 Použité technologie

V souvislosti s požadavky na systém a se získanými informacemi z dotazníku byly pro implementaci systému vybrány tyto technologie:

- JavaScript,
- HTML,

- CSS,
- framework Vue.js.

Podrobný popis všech technologií použitých v systému lze najít v práci Jaroslava Veselého [1]. Já se ve své práci zabývám pouze těmi hlavními.

2.4.1 JavaScript

JavaScript je jedním z nejpopulárnějších programovacích jazyků na světě, primárně je používaný k přidání automatizace, animace a interaktivity k webovým stránkám. Weboví vývojáři používají JavaScript na cokoliv od automatizace jednotlivých úkolů až po tvorbu komplexních webových stránek, které se chovají jako desktopové aplikace [3].

JavaScript používaný na webových stránkách je „client-side“ programovací jazyk. Což znamená, že scripty JavaScriptu se čtou, interpretují a spouštějí u klienta, což je váš webový prohlížeč. Pro porovnání, „server-side“ programovací jazyky běží na vzdáleném počítači, jako je například server hostující webovou stránku. Díky tomu, že je JavaScript „client-side“, může vývojář přidávat interaktivní prvky, které mění a aktualizují webovou stránku, aniž by musely načítat novou kopii stránky z webu [3].

Díky tomu, že náš systém bude obsahovat graf, který bude nutné často aktualizovat/překreslovat, je JavaScript skvělou volbou právě díky tomu, že běží přímo u klienta. Technologie, které běží na serveru, by mohly mít problém se zátěží při posílaní velkého množství dat.

2.4.2 HTML

HTML je zkratka pro HyperText Markup Language. HTML se používá na tvorbu elektronických dokumentů nazývaných stránky, které jsou dostupné na internetu. Každá stránka obsahuje sérii spojů na stránky ostatní. Těmto spojům se říká „hyperlinks“. Každá webová stránka, kterou na internetu vidíte, je napsána v jedné z verzí HTML [4].

HTML kód zajišťuje správné formátování textu a obrázků tak, aby je váš prohlížeč zobrazil tak, jak mají vypadat. HTML poskytuje základní strukturu stránky, nad kterou se poté pomocí CSS staví samotný vzhled stránky. HTML si lze tedy představit jako kostru (strukturu) webové stránky a CSS jako její kůži (vzhled) [4].

HTML se skládá z částí, kterým se říká elementy nebo také tagy. Tagem je například:

- tlačítko,
- pole na zadávání,
- popisek,

- atd.

Každá část stránky je tedy tagem nebo skupinou tagů.

2.4.3 CSS

CSS je zkratka pro „Cascading Style Sheets“ a jedná se o preferovanou cestu nastavování vzhledu stránky. Styly definují vzhled a pozici textů nebo jiných HTML tagů, zatímco HTML soubory definují jejich obsah a rozmístění. Jejich oddělení dovoluje měnit vzhled stránky, aniž byste ji museli celou přepisovat [5].

Slovo „cascading“ znamená, že změna provedená na rodičovskou komponentu se promítne i do všech jejích dětí. Například změníme-li barvu textu v *body*, všechny nadpisy a paragrafy v tomto tagu budou mít danou barvu [5].

Pravidla CSS jsou formovány z:

- Skupiny vlastností s hodnotami nastavenými tak, jak se má HTML obsah zobrazit. Například: „Chci, aby šířka tohoto elementu byla 50 % šířky rodiče a jeho pozadí bylo červené. “
- Selektorů, vybírajících elementy, na které se mají dané vlastnosti aplikovat. Například: „Chci vlastnosti aplikovat na všechny paragrafy v mém HTML dokumentu.“

Skupina CSS pravidel definuje, jak bude daná stránka vypadat [6].

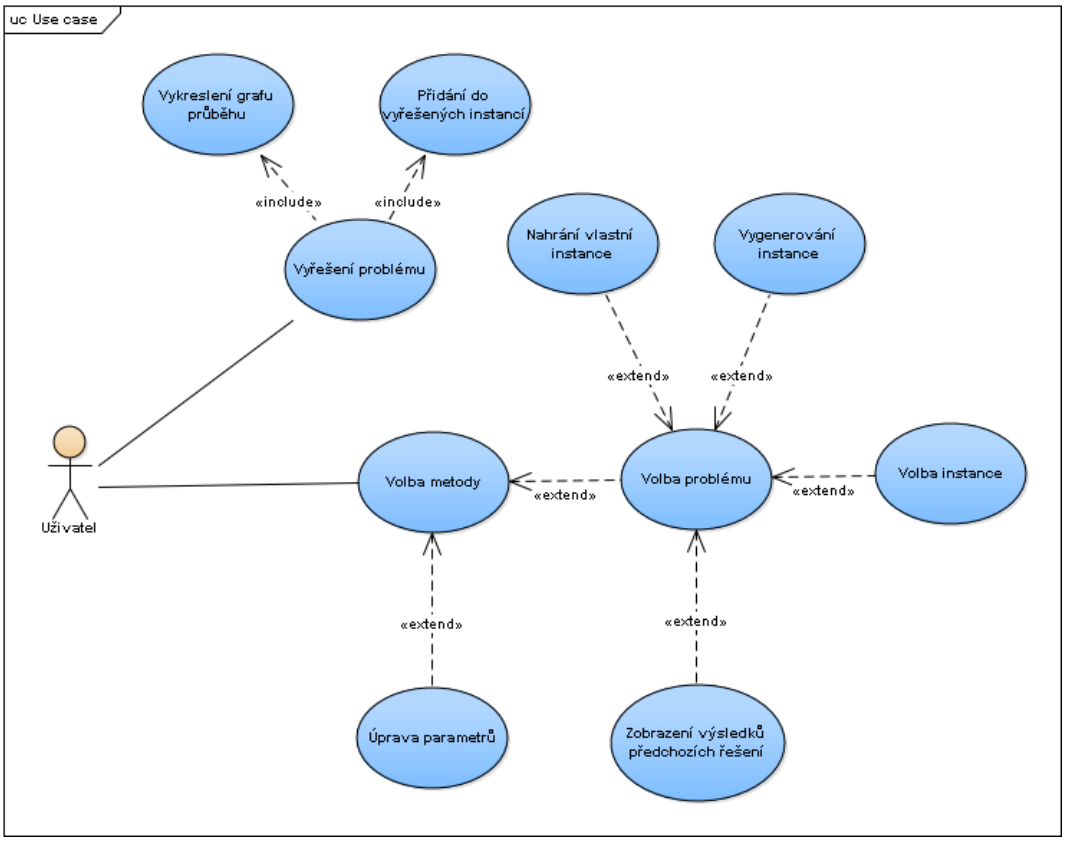
2.4.4 Vue.js

Po výběru jazyka, ve kterém bude systém implementovaný, bylo třeba zvolit framework, který nám při práci pomůže. Vzhledem k velikosti vyvíjeného systému jsme se nechtěli pouštět do velkých frameworků, jako jsou React nebo Angular, jejichž funkcionalitu bychom jen těžko využili a jejichž složitost by mohla být spíše na škodu než k užitku.

Z těchto důvodů jsme se rozhodli pro menší framework, který nás nezahltí funkcemi, ale i tak pomůže. Volba padla na framework Vue.js.

Vue (vyslovováno /vju:/, jako anglické slovo view) je progresivní framework na budování uživatelského rozhraní. Na rozdíl od ostatních jednolitých frameworků je Vue navrženo od začátku tak, aby bylo adaptivní. Jádro knihovny je zaměřeno pouze na pohledovou vrstvu a je jednoduché na pochopení a propojení s dalšími knihovnami nebo existujícími projekty. Na druhou stranu je Vue perfektně schopné pohánět sofistikované jednostránkové aplikace při spojení s moderními nástroji a podpůrnými knihovnami [7].

Hlavním důvodem této volby byla tedy jednoduchost, ale také znalost tohoto frameworku Jaroslavem Veselým, který se zabýval implementací uživatelského rozhraní.



Obrázek 2.5: Use case diagram

2.5 Analytická dokumentace

V této části práce se podíváme na analytickou dokumentaci systému, při které využiji UML diagramů.

2.5.1 Případy užití

Diagram případů užití neboli Use case diagram slouží k přehlednému popisu funkcionalit systému. U každého případu užití uvedu krátký popis a u vybraných připadů také scénář s ním spojený. Diagram viz obrázek 2.5.

Vyřešení problému

Hlavní z funkcionalit celého systému. Vyřeší zvolenou instanci vybranou metodou a zobrazí průběh řešení v grafu.

1. Případ užití začíná, když chce uživatel vyřešit instanci problému pomocí libovolné metody.
2. Uživatel zvolí jednu ze tří metod.
3. Zvolí problém, jehož instanci chce řešit.
4. Vybere instanci, kterou sám nahrál nebo vygeneroval pomocí generátoru.
5. U vybrané metody nastaví jednotlivé parametry.
6. Spustí řešení.
7. Include (Vykreslení grafu průběhu).
8. Include (Přidání do vyřešených instancí).
9. Zobrazení výsledků.

Vykreslení grafu průběhu

Systém během výpočtu vykresluje do grafu průběh řešení. Řešením je myšlena cenová funkce aktuální konfigurace.

Volba metody

Uživatel si může zvolit jednu ze tří nabízených metod:

- simulované ochlazování,
- genetický algoritmus,
- Tabu prohledávání.

Podle zvolené metody je pak možné měnit její specifické parametry.

Úprava parametrů

U každé ze tří metod si může uživatel libovolně nastavovat všechny parametry s ní spojené.

Volba problému

Po zvolení některé ze tří metod si uživatel může vybrat jeden z nabízených problémů, který chce pomocí zvolené metody řešit.

Nahrání vlastní instance

Uživatel má možnost nahrát do systému vlastní instanci problému, kterou chce řešit. Tato instance musí splňovat formát specifický pro tento typ problému.

Vygenerování instance

Systém umožňuje vygenerovat náhodnou instanci specifického problému na základě zvolených parametrů. Tyto parametry jsou závislé na typu problému.

1. Případ užití začíná, když si chce uživatel vygenerovat náhodnou instanci problému.
2. Uživatel otevře okno generátoru instancí.
3. Zvolí si problém, jehož instanci chce generovat.
4. Vyplní jednotlivé parametry generované instance.
5. Vygenerování instance.
6. Přidání instance do seznamu instancí daného problému.

Zobrazení výsledků předchozích řešení

Systém umožňuje zobrazit výsledky dříve řešených instancích. Tato řešení jsou zobrazena v panelu historie.

Volba instance

Umožňuje zvolit konkrétní instanci vybraného problému. Tato instance pak bude řešena při spuštění.

2.5.2 Diagramy aktivit

Diagramy aktivit slouží k bližšímu popisu případu užití. Popisují celkový průběh celého případu za pomoci jednotlivých kroků.

Vzhledem k jednoduchosti průběhu většiny případů užití jsem vytvořil diagramy aktivit pouze pro tyto případy:

- vyřešení problému,
- vygenerování instance.

Pro pohodlí uživatelů jsou všechna z uvedených nastavení (voleb) nastavena na defaultní hodnoty, takže je uživatel nemusí nastavovat ručně. V těchto případech je však uvádím jakožto individuální průchod, kdy chce uživatel nastavit vše podle sebe.

Vyřešení problému

Vyřešení problému je hlavní funkcionalitou celého systému a nejzajímavějším případem užití z hlediska počtu jednotlivých kroků.

Celý průběh je víceméně lineární, dělí se pouze v části výběru instance, kde má uživatel následující možnosti:

- **Výběr existující instance** – Uživatel si pro řešení vybere jednu z již existujících instancí, tedy instanci dříve nahranou nebo vygenerovanou.
- **Nahrání vlastní instance** – Tato volba umožňuje nahrát do systému instanci uživatele, uloženou v počítači nebo na jiném médiu. Je třeba, aby instance splňovala specifický formát problému, jehož instanci chceme nahrát.
- **Vygenerování instance pomocí generátoru** – Touto cestou si může uživatel vygenerovat instanci daného problému pomocí generátoru. Parametry instance si uživatel zvolí při generování.

Diagram viz obrázek 2.6.

Vygenerování instance

Druhý diagram popisuje průběh případu užití „Vygenerování instance“. Jedná se o jednoduchý průchod, který se dělí pouze v případě zadaní nevalidních parametrů. Uživatel zde má také možnost generovat více instancí, aniž by musel generátor znovu otevírat. Diagram viz obrázek 2.7.

2.5.3 Doménový model

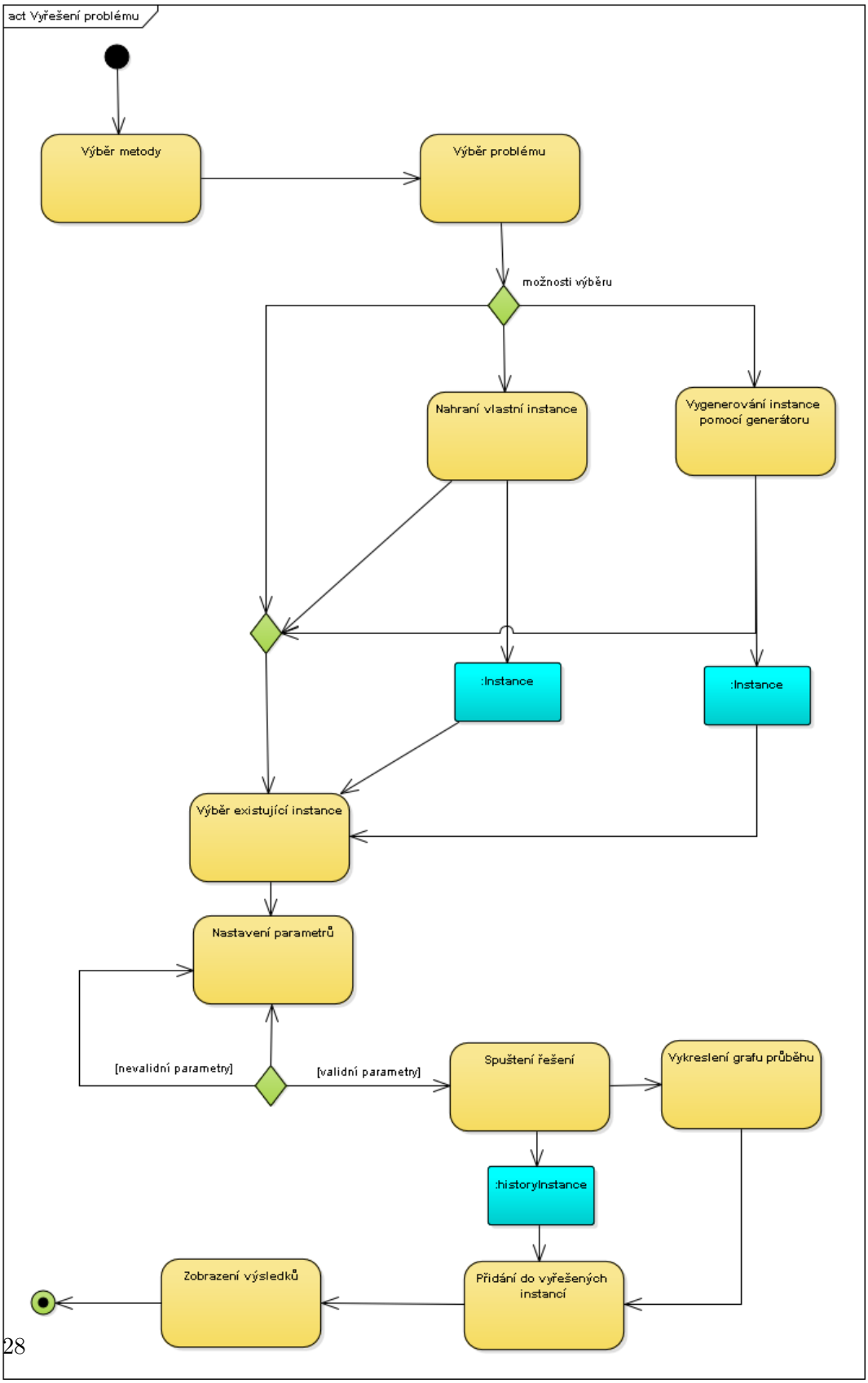
Doménový model slouží k popisu hlavních komponent (entit) systému. Nejedná se o obraz implementace, ale spíše o logický pohled na strukturu systému. Také se zde zaměřuji pouze na komponenty, které mají na starost logiku systému, nikoliv na třídy podpůrné, jako jsou:

- výsledek řešení,
- třídy starající se o běh výpočtu na pozadí (HTML 5 Web Workers),
- třídy na ukládání dat,
- a další.

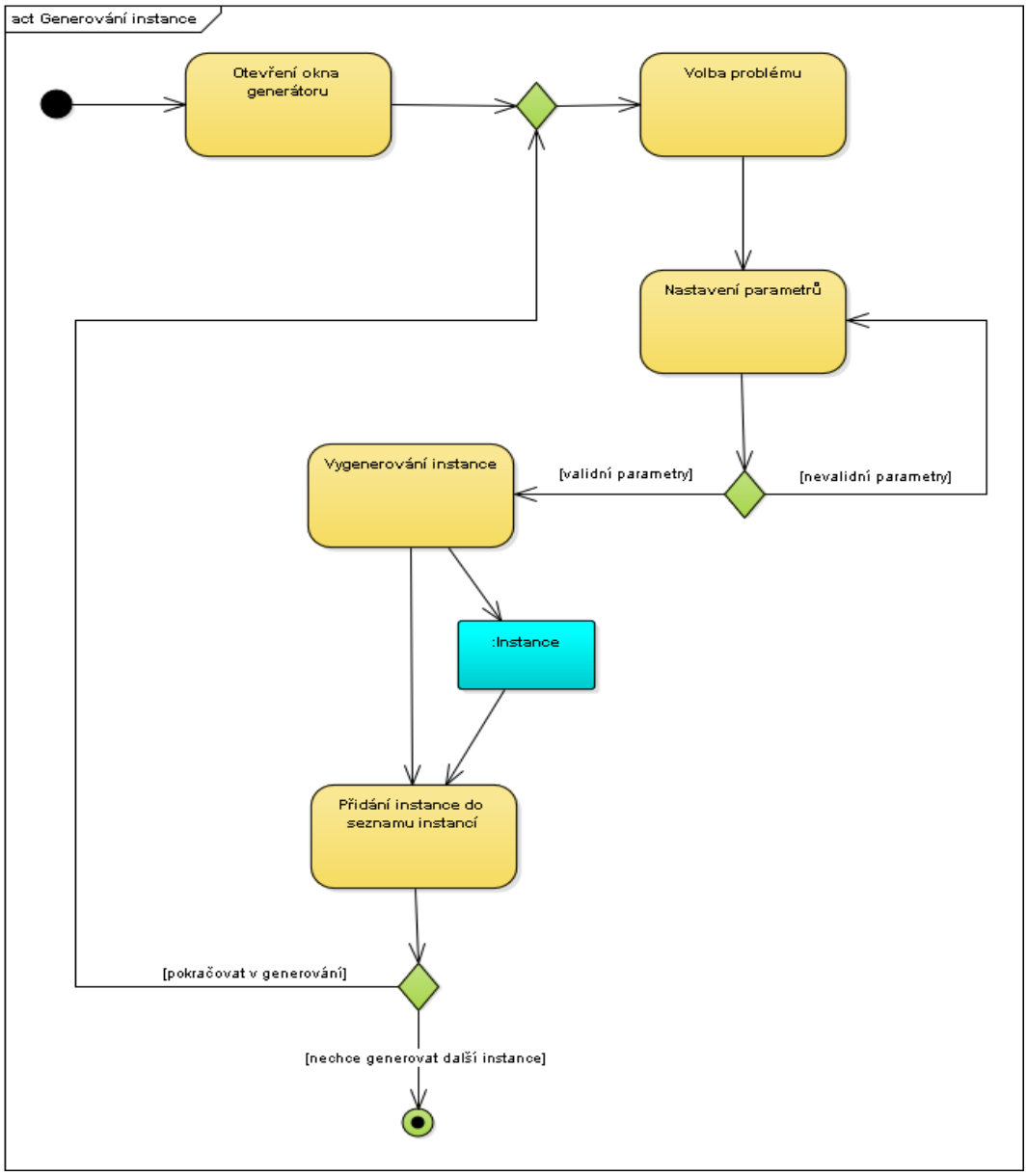
Bližší popis těchto komponent naleznete v práci Aplikace pro demonstraci funkce Tabu prohledávání [1].

Hlavními logickými entitami systému jsou tyto 4:

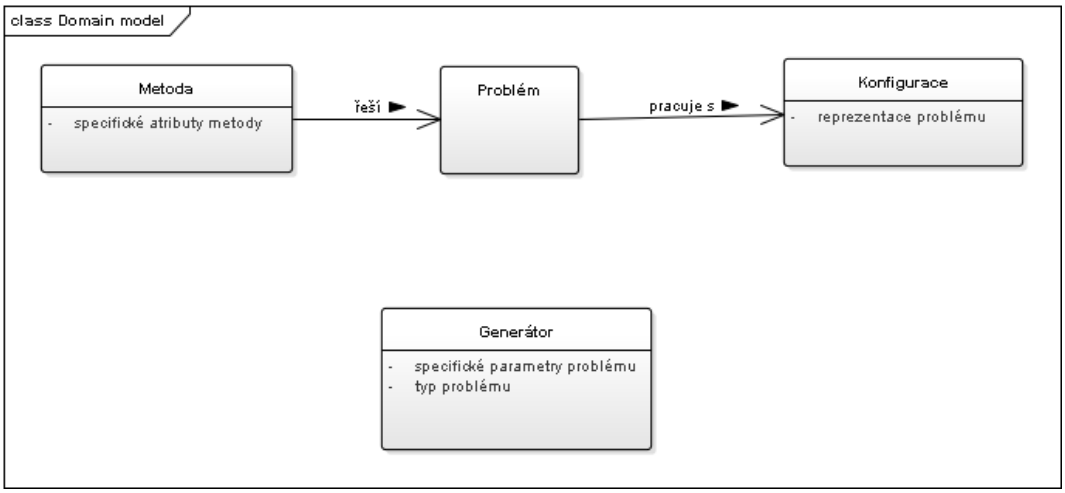
- metoda,



Obrázek 2.6: Vyřešení problému – diagram aktivit



Obrázek 2.7: Vygenerování instance – diagram aktivit



Obrázek 2.8: Doménový model

- problém,
- konfigurace,
- generátor.

V této části rozeberu všechny z těchto entit. Vysvětlím jejich význam v systému a také vysvětlím, jak jsou propojeny.

Doménový model lze vidět na obrázku 2.8.

Metoda

Metoda je hlavní entita systému. Jejím úkolem je řešit zadané problémy. Jinými slovy, jedná se o komponentu, která jako vstup dostává problémy a vrací jejich řešení.

Systém obsahuje tři metody:

- simulované ochlazování,
- Tabu prohledávání,
- genetický algoritmus.

Společným znakem všech metod je funkce, která dostává jako vstup problém a vrací výsledek. Každá metoda má také specifické parametry, které nastavuje uživatel, čímž ovlivňuje její průběh. Každá z metod je navržena tak, aby byla schopná řešit vícero typů problémů. Implementace metody musí být tedy nezávislá na typu problému a naopak.

Problém

Problém je entita, která se stará o popis specifických problémů. Udržuje tedy informace o řešené instanci[1] problému a obsahuje funkce pro výpočty hodnot nutných pro její vyřešení. Tyto funkce pak využívá entita *Metoda* k vyřešení daného problému.

Hlavní z těchto funkcí je funkce na výpočet cenové funkce. Cenová funkce se počítá pro jednotlivé konfigurace, které jsou další entitou systému. Cenová funkce je hodnota, která vyjadřuje, jak moc dobrá je daná konfigurace, tedy jak dobré řešení daná konfigurace reprezentuje.

Systém obsahuje tyto problémy:

- problém batohu,
- SAT problém,
- problém obchodního cestujícího,
- euklidovský problém obchodního cestujícího,
- problém minimálního uzlového pokrytí.

Konfigurace

Konfigurace je reprezentace řešení instance problému. Nejlépe ji lze vysvětlit na příkladu: Mějme problém batohu. Řešením tohoto problému je informace, které věci jsou v batohu a které ne. Konfigurací v tomto případě může tedy být bitové pole, kde index reprezentuje číslo věci a hodnota na tomto indexu pak značí, jestli věc v batohu je (hodnota 1), nebo není (hodnota 0).

Jedná se tedy o jakési kódování řešení do snadno zpracovatelného formátu. Problémy tedy umožňují mít vícero možných reprezentací konfigurací a zároveň vícero problémů může používat stejnou konfiguraci. Proto jsou tyto dvě entity od sebe odděleny, i když spolu úzce souvisí.

Shrnutí

Než se podíváme na poslední entitu, kterou je *generátor*, rád bych shrnul tři předchozí, protože spolu souvisí a jsou pro pochopení celé architektury systému klíčové.

Hlavní entitou je tedy *Metoda*, která pracuje s *Problémem*. *Metoda* obsahuje základní kostru iterativního algoritmu, která funguje nezávisle na typu problému. *Problém* tedy musí obsahovat funkce, které *Metoda* využívá k vyřešení instance daného problému. Hlavní takovouto funkcí je funkce na výpočet cenové funkce, což je hodnota určující kvalitu současného řešení, které je reprezentováno entitou *Konfigurace*. *Konfigurace* je současné řešení, kterému je

[1] Instance je jedno konkrétní zadání daného problému.

problém schopný přiřadit cenovou funkci, kterou využívá *Metoda* pro nalezení lepšího řešení.

Hlavní výhodou tohoto rozdělení je nezávislost metody na vstupním problému. Oddělení konfigurace nám umožňuje použít stejnou konfiguraci pro vícero problémů. Bohužel není možné, aby problém pracoval s vícero konfiguracemi, tedy není to možné bez vytvoření funkcí pro každou jednotlivou konfiguraci, protože výpočet cenové funkce je úzce spjatý právě s podobou konfigurace. Dále není možné počítat cenovou funkci bez problému (tedy v konfiguraci), protože výpočet závisí na vstupní instanci, kterou metoda řeší. Všechny tyto věci jsou důvodem pro toto rozdělení.

Bližší popis jednotlivých entit z pohledu implementace bude v následující kapitole, která bude o diagramu tříd.

Generátor

Generátor je entita, jejíž úkolem je generovat náhodné instance jednotlivých problémů na základě uživatelem zvolených parametrů. Každý problém má vlastní generátor s jinými parametry.

Generátor je oddělený od ostatních entit a funguje samostatně.

2.5.4 Diagram tříd

V této části rozeberu systém z pohledu implementace. Využiji k tomu diagramu tříd, viz obrázek 2.9, který však nebude obrazem implementace jedna ku jedné, ale spíše doménovým modelem rozšířeným o jednotlivé funkce. Popíši tedy rozhraní jednotlivých komponent systému, co jaké funkce dělají, jejich vstupy a výstupy. Všechny zde zmíněné funkce jsou klíčové pro fungování systému. Kompletní dokumentaci naleznete v příloze.

2.5.4.1 Method

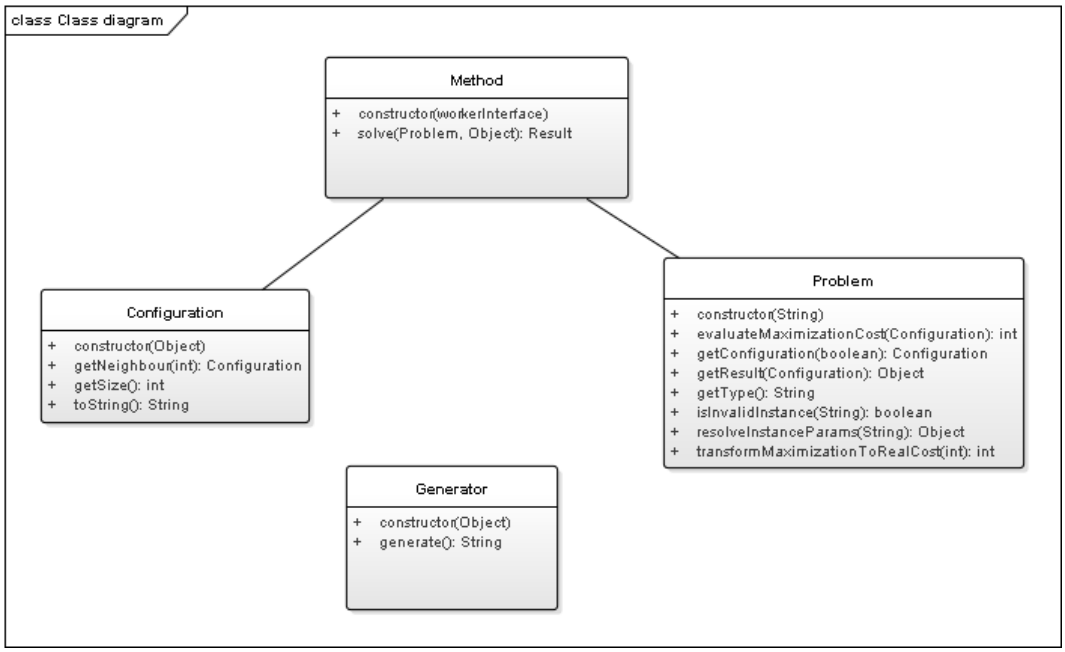
Jak už bylo zmíněno v kapitole 2.5.3, jedná se o třídu, jejímž úkolem je řešení problémů. Rozhraní třídy tohoto typu musí obsahovat pouze jedinou funkci a tou je funkce *solve*.

Solve Funkce, která vyřeší problém danou metodou.

Vstupní parametry:

- problem – třída typu problém, který chceme řešit. Může se jednat o jakýkoliv z možných typů problémů,
- params – parametry metody nastavené uživatelem. Jedná se o objekt obsahující všechny z těchto parametrů.

Výstupní parametry:



Obrázek 2.9: Diagram tříd

- result – jedná se o třídu typu *Result*, což je pomocná třída reprezentující výsledek řešení. Obsahuje tři parametry:
  - result – samotný výsledek, tedy nalezené řešení,
  - cost – cena/fitness tohoto řešení,
  - counter – počet provedených iterací k nalezení tohoto řešení.

constructor Konstruktor třídy dostává jako vstupní parametr instanci třídy workerInterface, což je pomocná třída, která se stará o správu Workerů[2].

2.5.4.2 Problem

Problem je třída, která se stará o reprezentaci problémů řešených třídou *method*.

getConfiguration Funkce vracející instanci třídy typu *Configuration*, se kterou daný problém pracuje.

Vstupní parametry:

[2]Worker je skript běžící na pozadí, aniž by zatěžoval samotnou stránku. Přesněji se jedná o HTML5 Web Workers.

- random – parametr typu Boolean, který určuje, zda bude vracená konfigurace náhodná, nebo ne.

Výstupní parametry:

- configuration – instance třídy Configuration. Jedná se o konfiguraci, kterou daný problém používá.

evaluateMaximizationCost Vrací cenovou funkci konkrétní konfigurace.
Vstupní parametry:

- configuration – konfigurace, jejíž cenovou funkci chceme spočítat.

Výstupní parametry:

- maximizationCost – hodnota cenové funkce vstupní konfigurace.

transformMaximizationToRealCost Vrací hodnotu optimalizačního kritéria z cenové funkce. Tato hodnota se používá pro výpis do grafu.
Vstupní parametry:

- maxCost – cenová funkce konfigurace.

Výstupní parametry:

- value – hodnota optimalizačního kritéria ze vstupní cenové funkce.

getType Vrací typ problému. V současné době obsahuje systém dva typy: permutační a binární. Pro genetický algoritmus je třeba tyto typy rozlišovat.
Vstupní parametry: Tato funkce nemá vstupní parametry.
Výstupní parametry:

- problemType – typ problému, jako String (enum).

getResult Vrací výslednou podobu konfigurace. Ve většině případů se jedná o samotnou konfiguraci, například o bitové pole. U obchodního cestujícího může však být výsledkem i opravdová cesta, což není přímo konfigurace.
Vstupní parametry:

- configuration – konfigurace, jejíž výslednou podobu chceme.

Výstupní parametry:

- result – objekt reprezentující výslednou podobu řešení.

isInvalidInstance Funkce, která zjišťuje, zda je formát instance v pořádku.
Vstupní parametry:

- instanceContent – kontrolovaná instance jako String.

Výstupní parametry:

- isInvalid – je-li instance nevalidní, nebo validní.

resolveInstanceParams Funkce vracející objekt s parametry vložené instance.
Vstupní parametry:

- instanceContent – instance, jejíž parametry chceme, jako String.

Výstupní parametry:

- instanceParams – objekt obsahující parametry vstupní instance.

constructor Konstruktor má pouze jeden parametr, jímž je instance problému v podobě Stringu.

2.5.4.3 Configuration

Configuration reprezentuje jednu konfiguraci daného problému v lehce zpracovatelné formě.

getNeighbour Nejdůležitější funkce třídy. Vrací sousední konfiguraci. Tedy stav, do kterého lze přejít jedním krokem ve stavovém prostoru.
Vstupní parametry:

- index – index, na kterém se má hodnota změnit. Tedy index, ve kterém se sousední konfigurace liší od této. Pokud není tato hodnota zadána, volí se náhodně. V případě permutačních konfigurací jsou vstupní indexy dva, protože sousední konfigurace se získá prohozením dvou prvků pole.

Výstupní parametry:

- configuration – instance třídy Configuration, která je sousedem konfigurace, na níž se funkce volala.

getSize Vrací velikost dané konfigurace. Tedy například velikost pole.
Vstupní parametry: Tato funkce nemá vstupní parametry.
Výstupní parametry:

- size – velikost konfigurace.

toString Vrací konfiguraci v tisknutelné formě.
Vstupní parametry: Tato funkce nemá vstupní parametry.
Výstupní parametry:

- string – konfigurace jako typ String.

constructor Konstruktorem konfigurace je objekt obsahující:

- informaci, zda má být vytvořena kopie na základě jiné konfigurace,
- velikost konfigurace (u batohu je velikost počet věcí, u SATu počet literálů atp.),
- Boolean, zda má být vytvořena konfigurace náhodně, nebo jako prázdná (u bitového pole například samé nuly).

2.5.4.4 Generator

Třída na generování instancí problémů.

generate Jediná funkce této třídy. Vrací instanci daného problému reprezentovanou jako *String*.
Vstupní parametry: Tato funkce nemá vstupní parametry.
Výstupní parametry:

- generatedInstance – instance problému reprezentovaná jako *String*.

constructor Konstruktor generátoru má jako parametr objekt obsahující parametry, které musí nová instance splňovat.

2.6 Návrh uživatelského rozhraní

V této části rozeberu návrh prototypu uživatelského rozhraní, čím jsem se při návrhu řídil a co bylo důvodem pro určitá rozhodnutí.

2.6.1 Nielsenova heuristická analýza

Nielsenova heuristická analýza je deset bodů, které by mělo uživatelské rozhraní splňovat. Používá se na hodnocení kvality uživatelského rozhraní, ale dá se použít již při návrhu jako šablona, kterou by měl výsledný návrh splňovat. Mezi těchto deset bodů patří:

- viditelnost stavu systému,
- shoda mezi systémem a realitou,
- minimální zodpovědnost,

- shoda s použitou platformou a obecnými standardy,
- prevence chyb,
- kouknu a vidím,
- flexibilita a efektivita,
- minimalita,
- smysluplné chybové hlášky,
- nápověda a dokumentace.

Dále se podíváme, co jednotlivé body znamenají.

Viditelnost stavu systému

Tento bod říká, že uživatel vždy musí vědět, co systém dělá. Tedy pokud se něco načítá, musí být vidět, že systém pracuje, ne aby vypadal zamrzlý, když ve skutečnosti není.

Shoda mezi systémem a realitou

Tlačítko s ikonkou koše znamená smazat, ikonka diskety zase uložit. Věci si musí zachovávat svůj význam, který je všeobecně známý.

Minimální zodpovědnost

Nedat uživateli možnost udělat věc, kterou není možné vrátit. Mít tlačítko zpět pro návrat do předchozího stavu/bodu. Pokud provádí akci, která je nenávratná, zeptat se uživatele, zda ji chce opravdu provést.

Shoda s použitou platformou a obecnými standardy

Systémy pro různé operační systémy by měly používat prvky, které jsou v daném systému standardně používané. Uživatel by měl vědět, co která věc znamená, tedy například nepoužívat zkratky, které člověk standardně nezná. To, co je napsáno, by opravdu mělo znamenat/dělat to, co říká.

Prevence chyb

Zabránit chybám dříve, než jsou provedeny. Například varovat uživatele před nesprávně zadanou hodnotou dříve, než ji potvrdí. Zvýrazňovat nesprávně vyplněná povinná pole atd.

Kouknu a vidím

Nenutit uživatele pamatovat si spoustu věcí. Vše, co uživatel potřebuje vědět, by měl mít na očích. Například pozice ve stromové struktuře (kde jsme na stránce / v jaké kategorii), počet kroků ve formuláři a další.

Flexibilita a efektivita

Tento bod je o rozdílu mezi běžným a zkušeným uživatelem. Není třeba mít běžně přístupná všechna možná nastavení a všechny funkce, protože běžný uživatel je nevyužije. Na druhou stranu je dobré, aby někde byly, a pokud je uživatel pokročilý, může je použít.

Minimalita

Nezobrazovat věci, které uživatel momentálně nepotřebuje. Čím méně toho na obrazovce je, tím lépe se v ní orientuje.

Smysluplné chybové hlášky

Veškeré chybové hlášky, které se mohou v systému vyskytnout, by měly být smysluplné. Není dobré vypisovat hlášky typu „java.lang.exception“. Místo toho řekněte uživateli, co se pokazilo a co může udělat, aby se to už nestalo. Například: „Rok narození musí obsahovat číselný údaj.“

Nápověda a dokumentace

Každý systém by měl obsahovat nějakou formu dokumentace nebo nápovědy, která uživateli pomůže při jeho používaní. Například:

- popis kolonek,
- forma vyplňované informace („Heslo musí být dlouhé alespoň pět znaků a obsahovat jedno číslo“),
- vysvětlivky zkratek,
- atd.

2.6.2 Wireframy

Wireframy jsou způsob, jak navrhnout webovou stránku na strukturální úrovni. Běžně se používají k rozložení obsahu a funkcionality na stránku v závislosti na potřebách uživatelů a uživatelských příbězích (tzv. „user stories“). Wireframy se používají v počátečních fázích vývoje, aby stanovily základní strukturu stránky před přidáním grafiky a obsahu [8].

Hlavní výhodou wireframů je poskytnutí brzké vizualizace, kterou lze projít s klientem. Uživatelé jej také mohou použít jako mechanizmus na dřívější testy použitelnosti. Nejen že jsou wireframy jednodušší na úpravy než konceptuální návrhy, ale také poskytují důvěru designerovi poté, co jsou schváleny klientem a uživateli [8].

V našem případě byly wireframy využity k domluvě na uživatelském rozhraní s dalšími kolegy, kteří na tomto systému pracují, ale také s vedoucím této práce. Na tvorbu wireframů jsem použil nástroj Balsamiq, který je dostupný online.

Navržené uživatelské rozhraní stoprocentně neodpovídá současné implementaci. Na provedené změny a důvody pro tyto změny se podíváme až v části realizace. Zde zmíním základní myšlenky spojené s tímto návrhem.

2.6.2.1 Hlavní strana

Návrh hlavní stránky viz 2.10. Cílem tohoto návrhu bylo vytvořit aplikaci tak, aby se v ní mohl uživatel volně pohybovat, aniž by se musel složitě vracet. Proto je celý systém navržený na jednu stránku, ze které má uživatel všechny potřebné věci snadno dostupné. V horní části okna jsou taby, pro každou metodu jeden. Po vybrání metody se zobrazí okno, které je pro každou metodu identické.

V levé straně okna je panel s parametry vybrané metody, pod nimi výběr problému a k němu seznam příslušných instancí. U výběru problému je také tlačítko, které slouží k nahrání nové instance nebo k otevření generátoru.

V pravé části je panel historie, kde jsou všechny dříve vypočtené instance problému, jejichž výsledky si může uživatel znovu zobrazit. Každá z vyřešených instancí se dá otevřít, což zobrazí parametry, s nimiž byla řešena. Tím se dostáváme k prostřední části okna, kde se zobrazuje průběh výpočtu a také vybrané výsledky z panelu historie. Toto okno má tři taby:

- Graf – zde se zobrazuje průběhu algoritmu.
- Stavový prostor – graf zobrazující průchod algoritmu stavovým prostorem.
- Výsledky – koncové výsledky algoritmu, jako je například:
  - nejlepší cena,
  - nejlepší řešení,
  - počet iterací,
  - čas běhu.

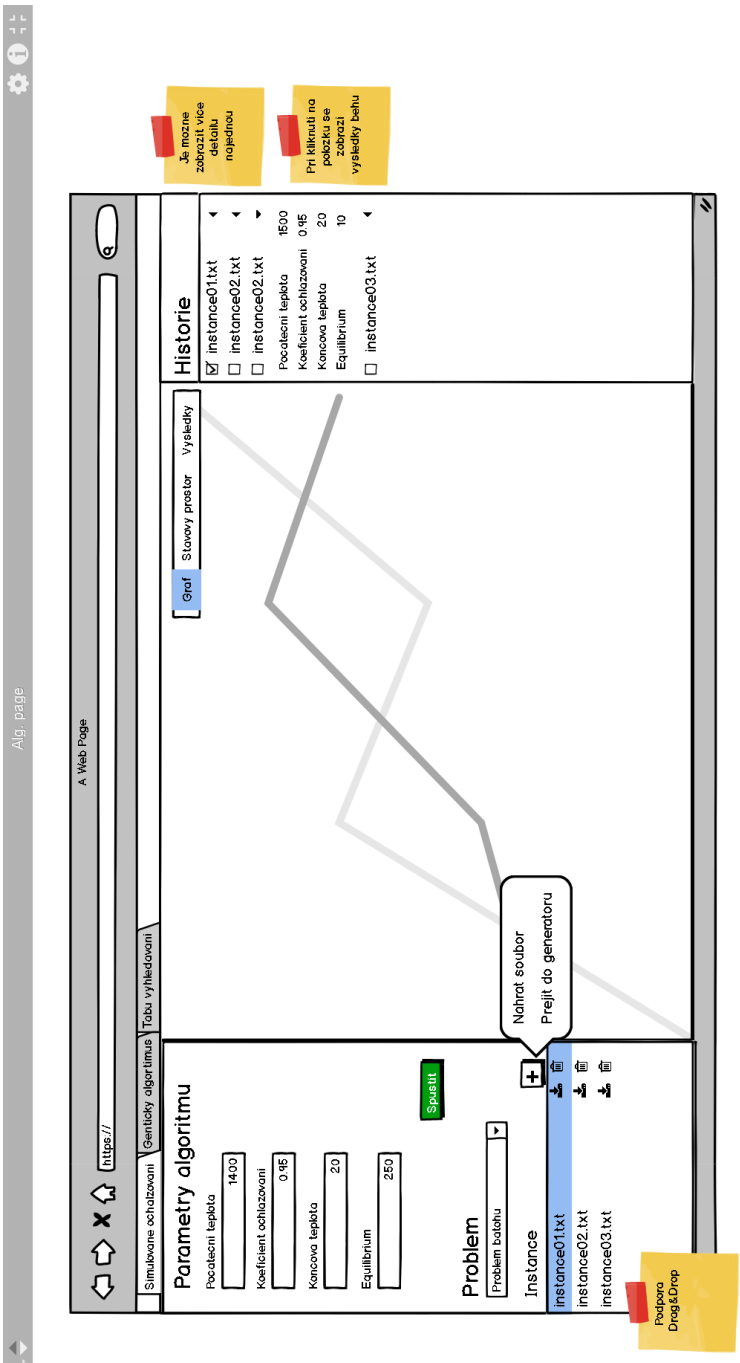
Při spuštění výpočtu se začnou do grafu vypisovat průběžné výsledky. Veškeré prvky jsou během výpočtu neaktivní až na tlačítko *spustit*, které se změní na *zastavit*.

2.6.2.2 Generátor

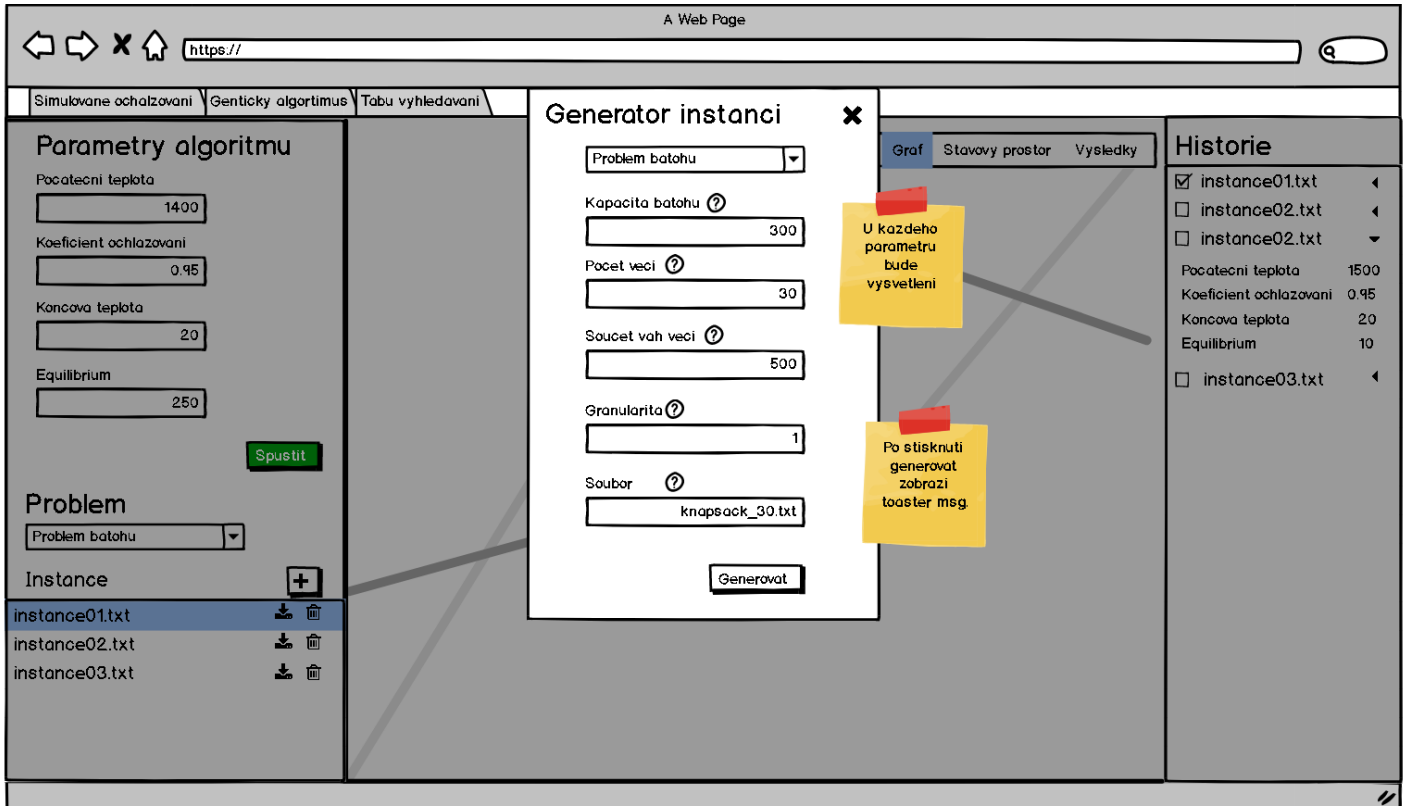
Interní generátor je řešený jako pop-up, který se zobrazí přes stránku po kliknutí na tlačítko generátoru. Samotný generátor je jednoduché okno, kde si uživatel nejdříve vybere problém, jehož instanci chce generovat, a poté nastaví parametry instance vybraného problému.

Tyto parametry jsou specifické pro každý problém, jediným společným je jméno vygenerované instance.

Po kliknutí na tlačítko generovat se zobrazí zpráva, že vygenerovaná instance byla přidána k seznamu instancí problému. Návrh je vidět na obrázku 2.11.



Obrázek 2.10: Návrh hlavní stránky



Obrázek 2.11: Návrh generátoru

Kapitola 3

Realizace

V této kapitole se blíže podíváme na metodu simulovaného ochlazování. Rozeberu implementované problémy a generátory instancí. Vysvětlím jak problematiku těchto věcí, tak svou implementaci. Dále zhodnotím současné uživatelské rozhraní a následně ho otestuji.

3.1 Simulované ochlazování

Tato část bude o simulovaném ochlazování. Vysvětlím, co to simulované ochlazování je, popíši implementaci této metody a také adaptační mechanizmus výpočtu počáteční teploty.

3.1.1 Co je to simulované ochlazování?

Simulované ochlazování je efektivní a obecná forma optimalizace. Je užitečné na hledání globálního optima v přítomnosti velkého množství lokálních optim. „Ochlazování“ odkazuje na analogii v termodynamice, především tím, jak se kovy ochlazují a žíhají. Simulované ochlazování však místo energie materiálu používá cenovou funkci optimalizačního problému [9].

Implementace simulovaného ochlazování je překvapivě jednoduchá. Algoritmus je v podstatě *hill-climbing* s tím rozdílem, že místo vybírání nejlepšího kroku vybírá krok náhodný. Pokud tento krok vylepšuje řešení, je vždy přijat. V opačném případě udělá algoritmus tento krok s pravděpodobností menší než jedna. Tato šance se exponenciálně zmenšuje se „špatností“ tohoto kroku, což je deltaE, o kterou bylo řešení zhoršeno (jinými slovy energie zvýšena) [9].

Pravděpodobnost přijetí zhoršujícího řešení je: $1 - e^{(deltaE/kT)}$ [9].

Parametr T se také používá při určování této pravděpodobnosti. Je analogický k teplotě žíhaného systému. Při větších hodnotách T se zhoršující kroky provádějí častěji. Čím více se teplota blíží k nule, tím jsou méně pravděpodobné a algoritmus se začíná chovat víceméně jako *hill-climbing*. V typické optimalizaci SA začíná T vysoko a je postupně snižováno podle „ochlazovacího

rozvrhu“. Parametr k je konstanta, která vztahuje relativní teplotu k energii (v přírodě se jedná o Boltzmannovu konstantu) [9].

Simulované ochlazování se typicky používá v diskrétních, ale velice rozsáhlých stavových prostorech, jako je množina možných průchodů měst v problému obchodního cestujícího a ve VLSI propojování[3]. Má rozsáhlou řadu aplikací, které jsou stále zkoumány [9].

3.1.2 Pseudokód

```
while(currentTemperature > endTemperature)
    for(var i = 0; i < innerCycleSize; i++)
        if(next better than current) current = next;
        else if(randomNumber < exp(-delta / currentTemperature))
        current = next

    currentTemperature *= coolingCoefficient
```

Z pseudokódu vidíme, že algoritmus je v základu celkem jednoduchý. V hlavním cyklu sledujeme, zda je současná teplota vyšší než *teplota minimální*. Je-li tato podmínka splněna, následuje vnitřní cyklus, který pro současnou teplotu projde počet stavů rovný hodnotě parametru *velikost vnitřního cyklu*. U každého ze stavů pak zjistí, zda je lepší než současný. Pokud ano, nastaví následující stav jako současný. Pokud ne, pak s pravděpodobností, která se odvíjí od současné teploty, buď tento stav přijme, což znamená, že ho nastaví jako stav současný, nebo zamítne. Pak následuje pouze snížení teploty v závislosti na *ochlazovacím koeficientu*.

Při rozhodování, zda je následující stav lepší než současný, využívá algoritmus cenovou funkci problému. Nejedná se tedy o funkci simulovaného ochlazování, ale o funkci, která je specifická pro daný problém, jak bylo vysvětleno v kapitole 2.5.3.

3.1.3 Parametry algoritmu

Algoritmus má 4 základní parametry:

- startovní teplota,
- minimální teplota,
- ochlazovací koeficient,
- velikost vnitřního cyklu.

Tyto 4 parametry je dobré znát pro pochopení fungování algoritmu.

[3] Skupina problému při propojování obvodů s velmi vysokou mírou integrace.

Startovní teplota

Startovní teplota nebo také počáteční teplota je parametr, který říká, s jakou teplotou bude algoritmus spuštěn. Tato teplota se odvíjí od cenové funkce problému a konkrétní řešené instance. Její nastavení je jedním z adaptačních mechanizmů, na které se podíváme později.

Obecně platí, že tato teplota musí být dostatečně vysoká, aby algoritmus přijímal i zhoršující kroky, ale ne moc vysoká, aby se na začátku „točil v kruhu“. Tedy přijímal moc zhoršujících stavů a nekonvergoval k nejlepšímu řešení. Příliš vysoká startovní teplota tedy nemá vliv na výsledek algoritmu, pouze prodlužuje jeho dobu běhu. Pokud však nastavíme startovní teplotu příliš nízkou, zvyšujeme pravděpodobnost uváznutí v lokálním optimu, protože algoritmus bude přijímat pouze lepší řešení a stane se z něj *hill-climbing*, což bylo již řečeno v předchozí kapitole.

Minimální teplota

Minimální teplota je hodnota, která říká, při dosažení jaké teploty se algoritmus ukončí. Pokud tedy byla *startovní teplota* teplota na začátku algoritmu, toto je teplota na jeho konci.

Co se týče nastavení této hodnoty, samozřejmě musí platit, že je menší než teplota startovní. Dále je nutné, aby nebyla moc vysoká, jinak se řešení dostatečně nepřiblíží k nejlepšímu řešení, protože víme, že se snižující se teplotou se snižuje i pravděpodobnost přijmutí zhoršujícího stavu, tudíž se algoritmus rychleji přibližuje k lepšímu řešení. Pokud nastavíme minimální teplotu příliš nízkou, pouze se prodlouží doba běhu, ale řešení to neovlivní. Což je logické, protože pokud řešení již konvergovalo k nejlepšímu možnému, kterého lze dosáhnout bez přijetí zhoršujícího stavu, nemůže algoritmus žádné další lepší řešení najít.

Stejně jako startovní teplota se tato hodnota odvíjí od cenové funkce řešeného problému a od konkrétní řešené instance.

Ochlazovací koeficient

Ochlazovací koeficient udává, o kolik se v každém cyklu sníží současná teplota. Ve své implementaci jsem použil exponenciální multiplikativní ochlazování, také známé jako geometrické ochlazování, které v každém ochlazovacím kroku provede toto: současná teplota * ochlazovací koeficient. Teplota v k-tém cyklu je tedy: $T_k = T_0 * \alpha^k$, kde α je ochlazovací koeficient. Tedy, čím více se tato hodnota blíží k jedné, tím více kroků algoritmus provede. A naopak čím více se blíží nule, tím rychleji se systém chladí, tudíž provede méně kroků.

Toto ochlazování jsem zvolil na základě článku A Comparison of Cooling Schedules for Simulated Annealing (Artificial Intelligence) [10]. V tomto článku se řeší, který ochlazovací postup je nejlepší. Právě exponenciální multiplikativní ochlazování vyšlo jako jedno z nejlepších a dle mého názoru se

jedná o nejjednodušší typ na nastavení, proto je to ideální volba do výukové aplikace.

Nelze říct přesný návod, jak tuto hodnotu volit, záleží na konkrétní instanci a na křivce průběhu výpočtu. Tato hodnota má na průběh podobný vliv jako *velikost vnitřního cyklu*, na kterou se podíváme nyní.

Velikost vnitřního cyklu

Jak už název napovídá, tato hodnota nastavuje velikost vnitřního cyklu. Což je cyklus, který se provádí uvnitř ochlazovacího cyklu a udává, kolik stavů algoritmus navštíví při současné teplotě.

Stejně jako *ochlazovací koeficient* zvyšuje nebo snižuje tato hodnota počet kroků, které algoritmus provede. Hodnota tohoto parametru by se měla odvíjet od velikosti řešené instance, kterou můžeme například vynásobit libovolnou konstantou.

3.1.4 Implementace

Náznak implementace jsem již zmínil v předchozí části. Nyní se podíváme na opravdovou implementaci, tedy na kód.

```
var currentTemp = +params.start_temp;
      var currentConfiguration = problem.getConfiguration();
      var currentCost = problem.evaluateMaximizationCost(
          currentConfiguration);
      var currentNeighbour = currentConfiguration.getNeighbour();


      // main cycle depending on temperature
      while (currentTemp > +params.min_temp) {
          //inner cycle
          for(var i = 0; i < +params.innerCycle; i++){
              var neighbourCost = problem.
                  evaluateMaximizationCost(currentNeighbour)
              //next is better
              if(currentCost < neighbourCost){
                  currentConfiguration = currentNeighbour;
                  currentCost = neighbourCost;
              }
              // next is worse
              else if(Math.random() < Math.exp((neighbourCost -
                  currentCost) / currentTemp)){
                      currentConfiguration = currentNeighbour;
                      currentCost = neighbourCost;
              }
              currentNeighbour = currentConfiguration.
                  getNeighbour();
              counter++;
          }
          currentTemp *= +params.cool_coef;
      }
```

Z kódu jsem odebral části, které se nestarají o logiku výpočtu, jako jsou: odesílaní dat do grafu, nastavování velikosti grafu atp.

Pokud porovnáme kód s pseudokódem, vidíme, že kostra je stále stejná a v principu se nic nezměnilo. Jediným rozdílem je pojmenování proměnných a doplnění o jednotlivé funkce.

Celý tento kód je funkce *solve* v metodě simulovaného ochlazování. U všech obsažených proměnných by mělo být jasné, co znamenají, s výjimkou *params*, což je objekt, který obsahuje všechny parametry simulovaného ochlazovaní získané z formuláře. A právě proto, že se jedná o hodnoty z formuláře, je třeba je „přetypovat“ na číselné hodnoty, což je řešeno znaménkem plus před touto proměnnou.

Veškeré funkce obsažené v kódu jsem vysvětlil v kapitole 2.5.4, proto je zde nebudu znovu zmiňovat.

Bez bližší znalosti funkce *evaluateMaximizationCost* by se mohlo zdát, že je v kódu problém. Tím problémem je znaménko „<“ při porovnání cenové funkce současného a následujícího stavu. Existují totiž dva typy problémů a to:

- minimalizující,
- maximalizující.

Minimalizující problémy se snaží cenovou funkci minimalizovat, zatímco maximalizující maximalizovat. Tudíž stav, který má cenovou funkci menší, je u minimalizujícího lepší a u maximalizujícího horší než stav s větší cenovou funkcí, což u porovnávaní za pomoci „<“ může způsobovat problémy. Tento problém však řeší samotná funkce *evaluateMaximizationCost*, která hodnotu cenové funkce vrací v takové podobě, aby toto porovnání vždy fungovalo a nebylo třeba funkci *solve* rozdělovat podle typu problému.

3.1.5 Adaptační mechanizmy

Ke kostře simulovaného ochlazování lze přidat další mechanizmy ovlivňující průběh. Mezi tyto mechanizmy patří například:

- výpočet počáteční teploty,
- výpočet velikosti vnitřního cyklu,
- restarty,
- zvyšování teploty,
- a další.

Já jsem ve své práci implementoval první dva zmíněné, tedy: výpočet počáteční teploty a velikosti vnitřního cyklu. Další z mechanizmů mohou být efektivní, ale svou povahou jdou proti principu simulovaného ochlazování.

3.1.5.1 Výpočet počáteční teploty

Výpočet počáteční teploty je adaptační mechanizmus, jehož úkolem je spočítat ideální startovní teplotu pro specifickou instanci daného problému. Existuje několik možností, jak tento problém řešit. Já se ve své práci rozhodl použít postup popsaný v práci Computing the Initial Temperature of Simulated Annealing [11].

Princip celé řešení je vcelku jednoduchý a obsahuje tyto kroky:

1. Zvol počáteční teplotu a požadovanou pravděpodobnost přijetí zhoršujícího stavu.
2. Vytvoř množinu stavů a ulož jejich energie.
3. Spočti pravděpodobnost přijetí těchto stavů při současné teplotě.
4. Pokud se tato pravděpodobnost rovná pravděpodobnosti přijetí, kterou chceme, tak se cyklus ukončí. Pokud ne, uprav teplotu na základě získané pravděpodobnosti a vrať se zpět na krok 3.

Krok 1: Zvolíme počet přijatých zhoršujících stavů χ_0 (stejné jako pravděpodobnost přijetí zhoršujícího stavu). Poté nastavíme počáteční teplotu T_0 na jakékoliv kladné číslo.

Krok 2: Generujeme náhodné stavy a ukládáme si množinu S, která obsahuje přechody mezi těmito stavy, a hodnotu maximální a minimální energie E tohoto přechodu. Vysvětleno na příkladu: Máme stav, který má energetickou hodnotu (což je cenová funkce) 10, poté vygenerujeme stav s energií 7. Tudíž do množiny přidáme přechod, který má maximální energii E_{max} 10 a minimální E_{min} 7. Takto vygenerujeme libovolně velkou množinu.

Krok 3: Spočteme pravděpodobnost χ_{T_n} přijetí stavu (což je opět i množství přijatých stavů) při současné teplotě. Tuto hodnotu spočteme pomocí vzorce:

$$\chi_T = \frac{\sum_{t \in S} \exp(E_{max_t}/T)}{\sum_{t \in S} \exp(E_{min_t}/T)}$$

Krok 4: Pokud se spočtená pravděpodobnost rovná námi zvolené pravděpodobnosti χ_0, algoritmus končí a současná hodnota teploty T_n je hledaná počáteční teplota. Pokud se nerovná, upravíme teplotu T_n podle vzorce:

$$T_{n+1} = T_n \frac{\ln(\chi_{T_n})^{(1/p)}}{\ln(\chi_{T_n})}$$

A poté pokračujeme zpět na krok číslo 3. p je ve vzorci libovolné reálné číslo větší nebo rovno jedné.

V mé implementaci je $\chi_0 = 0.5$, $p = 1$ a počáteční teplotu T_0 nastavuji na maximum ze všech energií. Hlavním důvodem pro tuto volbu T_0 je, že funkce *Math.exp()* vrací pro hodnoty od $\exp(-726)$ a nižší nulu. Což je poté samozřejmě problém, protože se může stát, že dělím nulou. Jakmile se začne teplota snižovat, může se samozřejmě opět stát, že funkce vrátí nulu, ale v této době již bude teplota blízko tomu, co hledáme, díky čemuž můžeme celý proces ukončit.

Co se týče velikosti množiny S, tu nastavuji na 200. Nastavit správně tuto hodnotu není však jednoduché, protože pokud vytvoříme moc velkou množinu, může se stát, že bude obsahovat spoustu přechodů s malým zhoršením, což povede k nastavení moc malé teploty. Pro malou hodnotu naopak generujeme značně rozdílené množiny, takže výsledná teplota se může při každém spuštění velmi lišit. Oba tyto problémy také závisí na samotné instanci, díky čemuž není možné tento problém vyřešit pro všechny případy.

3.1.5.2 Velikost vnitřního cyklu

Výpočet velikosti vnitřního cyklu je triviální funkce. Implementoval jsem ji pro pohodlí uživatelů tak, aby bylo možné nastavit velikost vnitřního cyklu na hodnotu, která pro danou instanci dává smysl.

Funkce pouze vezme velikost vybrané instance a její dvojnásobek nastaví jako velikost vnitřního cyklu. Jedná se o triviální, ale fungující řešení.

3.2 Problémy

Velmi důležitou částí systému jsou kromě jednotlivých iterativních metod také problémy, které metody řeší. Systém v současné době obsahuje 5 problémů:

- problém batohu,
- problém obchodního cestujícího,
- euklidovský problém obchodního cestujícího,
- problém minimálního uzlového pokrytí,
- SAT problém.

Ve své práci jsem se zabýval implementací problému obchodního cestujícího a problémem minimálního uzlového pokrytí. Problém batohu jsme implementovali společně s kolegy jakožto šablonu pro interface implementovaných problémů. Popis tohoto rozhraní lze najít v kapitole 2.5.4.

Aby bylo jasné, jak problémy fungují, je třeba vysvětlit konfigurace, se kterými problémy pracují, proto se na ně podíváme jako první. Systém obsahuje dva typy konfigurací:

- bitové pole,
- permutace.

3.2.1 Konfigurace

Účel a smysl konfigurace jsem již vysvětlil v kapitolách 2.5.3 a 2.5.4.

3.2.1.1 Bitové pole

Bitové pole je třída, která, jak už název napovídá, reprezentuje konfiguraci instance problému jako bitové pole.

Funkce pro získání souseda u bitového pole provede u konfigurace, jejíž souseda chceme, „flip“ bitu na náhodné nebo vybrané pozici a tuto nově získanou konfiguraci vrátí jako výsledek.

Funkce *getSize* vrací velikost bitového pole, které reprezentuje danou konfiguraci.

Tento typ konfigurace se používá u problémů:

- problém batohu,
- SAT problém,
- problém minimálního uzlového pokrytí.

3.2.1.2 Permutace

Tato třída reprezentuje konfiguraci jako pole, které obsahuje permutaci.

Získání souseda je v tomto případě prohození dvou pozic v poli. Pokud chceme konkrétního souseda, je třeba předat funkci dva indexy, které udávají pozice prvků, které se mají prohodit.

Tato konfigurace se používá u problémů obchodního cestujícího.

3.2.2 Problém batohu

Problém batohu je jedním z nejznámějších kombinatorických problémů. Zadání je jednoduché: Mějme batoh s kapacitou W a množinu věcí N. Každá z věcí množiny N má svoji váhu w a cenu v. Cílem tohoto problému je vložit do batohu věci z množiny N tak, aby součet vah w nepřekračoval kapacitu W a součet jejich cen v byl maximální.

Vstupem je tedy kapacita batohu a množina věcí. Výstupem je podmnožina množiny věcí, která splňuje, že součet vah věcí z této podmnožiny je menší nebo rovný kapacitě batohu. Optimalizačním kritériem je celková cena vybraných věcí, která má být co největší.

3.2.2.1 Možnosti řešení a složitost

Složitost problému batohu se odvíjí od způsobu řešení. Nejtriviálnějším přístupem je hrubá síla, kdy algoritmus vyzkouší všechny možné konfigurace batohu. Těchto konfigurací je $2^{|N|}$. Tuto složitost je možné zlepšit pomocí chytrého ořezávání, ale asymptoticky zůstává stále stejná.

Dalším přístupem je využít dynamického programování. Zde můžeme použít postupy:

- dekompozice podle ceny,
- dekompozice podle kapacity.

Oba tyto postupy mají pseudopolynomiální složitost, což znamená, že tato složitost se odvíjí od parametru, který nesouvisí s velikostí řešené instance. Pro dekompozice podle ceny je to celková cena všech věcí V a složitost je tedy $V * |N|$. Dekompozice podle kapacity má složitost $W * |N|$.

Aproximační třída problému batohu je FPTAS, tedy pro řešení existuje plně polynomiální aproximační schéma.

Další možností je samozřejmě řešení pomocí iterativní metody, na kterou se podíváme podrobněji.

3.2.2.2 Implementace

Jak už bylo řečeno v kapitole 2.5.4, nejdůležitější funkcí u třídy typu problém je funkce na výpočet cenové funkce. U problému batohu je cenovou funkcí celková cena věcí v batohu, což je přesně to, co tato funkce vrací.

Tento problém využívá konfiguraci bitové pole, kde sousedem je batoh, ze kterého se jedna věc odebrala, nebo naopak přidala.

Kód na výpočet cenové funkce:

```
evaluateMaximizationCost(bitArrayConfig) {
    if (bitArrayConfig === null) return -1;

    const bitArray = bitArrayConfig.getBitArray();

    var sumValue = 0;
    var sumWeight = 0;

    bitArray.forEach((bit, index) => {
        if (bit) {
            sumValue += this._items[index].value;
            sumWeight += this._items[index].weight;
        }
    });

    return (sumWeight > this._capacity) ? this._capacity -
        sumWeight : sumValue;
}
```

Jak v kódu vidíme, funkce pouze projde celé pole a spočte celkovou cenu a váhu všech věcí v batohu. Je-li celková váha menší nebo rovna kapacitě batohu, vrátí cenu všech věcí, pokud je větší, vrátí zápornou hodnotu, o kterou byla kapacita překročena. Vracet zápornou hodnotu se může zdát jako podivné řešení, ale je třeba, aby byla splněna podmínka na kapacitu. Tudíž konfigurace, která tuto podmínku nesplňuje, musí být vždy horší než ta, která ji splňuje.

3.2.3 Problém minimálního uzlového pokrytí

Problém minimálního uzlového pokrytí spočívá v nalezení nejmenší množiny uzlů, a to takové, že množina hran, které vedou z těchto uzlů, je rovna množině všech hran.

Vstupem problému je tedy libovolný neorientovaný graf a výstupem množina uzlů, která pokrývá všechny hrany. Optimalizačním kritériem je, aby tato množina byla co nejmenší.

Pro obecný graf je složitost tohoto problému (při řešení hrubou silou) rovna $2^{|V|}$. Existují však případy, které lze řešit v polynomiálním čase, a to v případě, že se jedná o bipartitní graf nebo o strom.

Aproximační třída tohoto problému je APX-úplný.

3.2.3.1 Implementace

Kód pro výpočet cenové funkce:

```
evaluateMaximizationCost(bitArrayConfig) {
    if (bitArrayConfig === null) return -1;

    const bitArray = bitArrayConfig.getBitArray();
    var numberOfCovered = 0;
    var currentPrice = 0;
    var edgeArray = this._array.map(x => x.slice());

    for(var i = 0; i < bitArray.length; i++)
    {
        if(bitArray[i]) {
            currentPrice++;
            for(var j = 0; j < this._size; j++)
            {
                if(edgeArray[i][j] === 1) {
                    numberOfCovered++;
                     // edge covered
                    edgeArray[i][j] = 2;
                    edgeArray[j][i] = 2;
                }
            }
        }
    }
```

```
return ((this._noEdges - numberOfCovered) === 0 ? this.
    _size - currentPrice :  numberOfCovered - this.
    _noEdges);
```

Funkce spočte, kolik uzlů bylo vybráno do množiny, která má pokrýt graf, a zároveň spočte, kolik hran bylo pokryto. Pokud jsou pokryty všechny hrany, vrátí funkce počet nevybraných uzlů. Důvod pro vrácení této hodnoty místo počtu vybraných uzlů byl vysvětlen v implementaci simulovaného ochlazování, kde algoritmus neřeší, zda se jedná o minimalizační, nebo maximalizační problém, a proto se musí všechny chovat jako maximalizační. Pokud nejsou pokryty všechny hrany, vrátí záporný počet nepokrytých hran. Stejně jako u problému batohu je tomu tak proto, že řešení, které nepokrylo celý graf, musí být horší než každé řešení, které ho pokryje.

3.2.4 Problém obchodního cestujícího

„Problém obchodního cestujícího je NP-těžký optimalizační problém, jehož zadání neformálně zní například takto: Je dáno několik měst a vzdálenosti mezi nimi. Úkolem je najít takovou cestu, která navštíví každé město právě jednou. Formálně se jedná o nalezení nejméně ohodnocené hamiltonovské kružnice v úplném ohodnocení grafu. Řešením úlohy je posloupnost měst, která udává, v jakém pořadí se mají města projet.“ [12]

„Problém je pojmenovaný podle obchodníků, kteří jezdili od města k městu a snažili se tam prodat co nejvíce vysavačů, psacích strojů nebo třeba encyklopedií. Snažili se proto svou cestu vybranými městy naplánovat tak, aby je všechna projeli s minimálními náklady, tedy po co nejkratší trase.“ [12]

Právě díky pojmenování tohoto problému jsem se ho rozhodl implementovat ve dvou formách. První forma opravdu řeší hledání nejkratší cesty při návštěvě vybraných měst. V této formě tedy umožňuji jako vstup jakýkoliv silně souvislý graf, protože reálně nevede cesta z každého města do každého města přímo, a v tomto grafu je dále možné vybrat pouze specifické uzly, které je třeba navštívit. Výstupem je opravdová cesta, kterou cestující musí jet, ne pouze pořadí měst, která musí navštívit. Vzhledem k tomu, že graf není úplný, může tento výstup obsahovat některé uzly vícekrát. Tento typ je označen jako „Shortest“.

Druhá forma je typickou variantou obchodního cestujícího, kde je cílem nalézt nejkratší hamiltonovskou kružnici v úplném grafu, proto jsem ji nazval „Hamiltonian“.

Aproximační třída složitosti tohoto problému je NPO-úplný.

3.2.4.1 Implementace

Oproti předchozím problémům je implementace u obchodního cestujícího složitější, obzvláště v případě formy hledající reálnou nejkratší cestu. Z tohoto

důvodu vezmu celou implementaci postupně a vysvětlím i ostatní funkce, ne pouze funkci na výpočet cenové funkce.

První důležitou funkcí pro fungování první formy problému je Floyd-Warshallův algoritmus pro výpočet nejkratších cest v grafu. Díky tomu, že vstupem může být jakýkoliv silně souvislý graf, je třeba spočítat nejkratší cesty z každého uzlu do každého uzlu, protože ne vždy mezi uzly existuje hrana, a i když existuje, nemusí to být nejkratší cesta. Implementace Floyd-Warshall algoritmu:

```
_floydWarshall() {
    for(var i = 0; i < this._noNodes; i ++)
    {
        for(var j = 0; j < this._noNodes; j ++)
        {
            for(var k~= 0; k~< this._noNodes; k~++)
            {
                if(this._distanceArray[j][k] > this.
                    _distanceArray[j][i] + this._distanceArray[i
                    ][k]) {
                    this._distanceArray[j][k] = this.
                        _distanceArray[j][i] + this.
                        _distanceArray[i][k];
                    this._pathArray[j][k] = this._pathArray[j][i
                        ];
                }
            }
        }
    }
}
```

Jedná se o jednoduchý algoritmus, který v čase $|V|^3$ spočte nejkratší cesty mezi všemi uzly. Proměnná *distanceArray* je pole, kam se ukládají délky těchto cest, a proměnná *pathArray* slouží k rekonstrukci všech těchto cest, k čemuž se dostaneme později.

Je-li počet uzlů, které se mají navštívit, menší než počet všech uzlů grafu, není třeba počítat nejkratší cestu mezi všemi uzly. V takovémto případě stačí spočítat cestu mezi uzly, které se mají navštívit, a všemi ostatními. V tomto případě je složitost $|A| * |V|^2$, kde A je množina uzlů, které se mají navštívit. Pro tento výpočet používám upravený Dijkstrův algoritmus:

```
_dijkstra() {
      var distanceArray = [];
      var pathArray = [];
      var nodes = [];
      var shortest;
      var chosenNode;

      const distances = this._distanceArray;
      // for all nodes
      for(var i = 0; i < this._noNodesToVisit; i++)
      {
        //initialization
        for(var j = 0; j < this._noNodes; j++)
```

```
    {
      distanceArray[j] = Number.MAX_SAFE_INTEGER;
      pathArray[j] = null;
      nodes.push(j);
    }

    distanceArray[this._nodesToVisit[i]] = 0;
    //while there are unvisited nodes
    while(nodes.length > 0)
    {
      shortest = Number.MAX_SAFE_INTEGER;
      for(var j = 0; j < nodes.length; j++)
      {
        if(distanceArray[nodes[j]] < shortest) {
          shortest = distanceArray[nodes[j]];
          chosenNode = nodes[j];
        }
      }
      //remove node with shortest path
      nodes.splice(nodes.indexOf(chosenNode), 1);
      // update shortest path and array for path rebuilding
      for(var j = 0; j < this._noNodes; j++)
      {
        if(shortest + distances[chosenNode][j] <
            distanceArray[j]) {
          distanceArray[j] = shortest + distances[chosenNode
              ][j];
          pathArray[j] = chosenNode;
        }
      }
    }
    pathArray[this._nodesToVisit[i]] = this._nodesToVisit[i];

    // rebuild all paths and update _pathArray and
        _distanceArray arrays
    var currentNode;
    for(var k~= 0; k~< pathArray.length; k++)
    {
      currentNode = k;
      while(pathArray[currentNode] !== this._nodesToVisit[i])
      {
        this._pathArray[pathArray[currentNode]][k] =
            currentNode;
        currentNode = pathArray[currentNode];
      }
      this._pathArray[this._nodesToVisit[i]][k] = currentNode
          ;
      this._distanceArray[this._nodesToVisit[i]][k] =
          distanceArray[k];
    }

  }

}
```

Další pomocnou funkcí je funkce, jejímž úkolem je „napojit“ permutaci na konkrétní problém. Co to znamená, se nejlépe vysvětlí na příkladu: Třída konfigurace uchovává pole, které má hodnoty číslované od nuly, takže například 0-3-1-2. Cílem řešené instance je navštívit uzly 6, 9, 13, 18. Tato funkce tedy vezme tato dvě pole a naváže permutaci na konkrétní řešení, tudíž vrátí pole 6-18-9-13, což je reálné pořadí, v jakém se mají města projet. Implementace této funkce:

```
_bindToNodes(permutationConfig) {
    var myPermutation = permutationConfig.map(x=>x);

    for(var i = 0; i < this._noNodesToVisit; i++)
    {
        myPermutation[i] = this._nodesToVisit[myPermutation[i]];
    }

    return myPermutation;
}
```

Dostáváme se k funkci na výpočet cenové funkce. Tato funkce vezme současnou konfiguraci průchodu měst a spočte cenu cesty. Implementace:

```
evaluateMaximizationCost(permutationConfig) {
    var price = 0;
    var permutation = permutationConfig.getArray();

    if(this._type === "Shortest") permutation = this._bindToNodes
        (permutation);

    for(var i = 0; i < permutation.length - 1; i++)
    {
        price -= this._distanceArray[permutation[i]][permutation[
            i+1]];
    }

    price -= this._distanceArray[permutation[permutation.length -
         1]][permutation[0]];

    return price;
}
```

Jak je vidět, funkce používá funkci *_bindToNodes*, kterou jsem vysvětlil dříve, a pole *distanceArray*, obsahující ceny nejkratších cest mezi uzly. Podmínka zajišťuje, že pokud jde o typický problém obchodního cestujícího, počítá se hodnota cenové funkce pro získanou permutaci, protože v tomto případě se jedná pouze o průchod všech měst, která jsou číslovaná stejně jako samotná konfigurace.

Funkce vrací zápornou hodnotu délky této cesty opět proto, aby se problém choval jako maximalizační.

Poslední funkce, kterou je třeba vysvětlit, je funkce na sestavení reálné cesty z pořadí měst, která se mají navštívit. Funkce využívá pole pathArray,

které obsahuje nejkratší cesty mezi uzly. Díky čemuž vrátím opravdovou cestu grafu, nikoliv jen pořadí měst, která se musí navštívit. Implementace této funkce:

```
rebuildPath(permutationConfig) {
    var permutation = this._bindToNodes(permutationConfig);
    var path = [permutation[0]];

    for(var i = 1; i < permutation.length; i++)
    {
        while(permutation[i-1] != permutation[i])
        {
            permutation[i-1] = this._pathArray[permutation[i-1]][
                permutation[i]];
            path.push(permutation[i-1]);
        }
    }
    path.push(path[0]);

    return path;
}
```

Pokud se jedná o typický problém obchodního cestujícího, vrací se pouze pořadí průchodů uzlů, tedy konfigurace.

3.3 Generátory

Posledním typem třídy, o které budu mluvit, je třída na generování instancí problémů. Ke každému problému existuje jeden generátor. Ve své práci jsem se zabýval implementací generátorů k těmto problémům:

- problém batohu,
- SAT problém,
- problém obchodního cestujícího,
- problém minimálního pokrytí.

Hlavním důvodem pro implementaci těchto tříd bylo, aby měl uživatel možnost testovat metody na různých instancích, aniž by si je musel ručně generovat.

3.3.1 Problém batohu

Generátor problému batohu obsahuje tyto parametry:

- počet věcí,
- kapacita batohu,

- sumární váha věcí,
- maximální hodnota věci,
- granularita.

Splnit většinu těchto parametrů je velice jednoduché. Jedinými složitějšími parametry jsou sumární váha a granularita.

Pro dodržení parametru sumární váhy používám jednoduchý postup, kdy nejdříve vytvořím náhodně váhy všech věcí. Tyto váhy musí splňovat jedinou podmínku, a to tu, aby nepřekračovaly sumární váhu. Důvodem pro tuto podmínku je to, abych omezil horní hranici číslem a zároveň umožnil vygenerování váhy věci až do maxima sumární váhy (ve výsledku bude vždy nižší, protože ostatní věci taky něco váží). Abych poté dosáhl toho, že součet vah je uživatelem zadaná hodnota, jednoduše všechny tyto váhy vydělím výrazem $\frac{soucasnaSumarniVaha}{sumarniVaha}$. Vzniká zde však problém při zaokrouhlování, kdy nová sumární váha nemusí o malý zlomek sedět. Tento problém se však snadno vyřeší přičtením tohoto rozdílu k náhodně vybrané váze.

Druhým parametrem, který je třeba řešit, je granularita. Granularita udává, jaký je podíl těžkých a lehkých věcích v batohu. Pro hodnoty větší než nula bude batoh obsahovat spíše těžké věci, pro hodnoty menší než nula spíše věci lehké. Tento problém řeší jednoduchý vzorec, který jsem převzal z generátoru instancí předmětu MI-PAA, kdy pro granularitu větší než nula má věc s váhou n následující šanci, že se dostane do batohu:

$$\frac{1}{(sumarniVaha - n)^{granularita}}$$

Je-li granularita menší než nula, chceme-li tedy spíše lehké věci, použije se tento vzorec:

$$\frac{1}{(n)^{-granularita}}$$

S tímto přístupem však vzniká také několik problémů. Prvním problémem je, že pokud bude granularita moc velká / malá, spousta vah bude zamítnuta, tudíž generování bude trvat o poznání déle.

Druhý problém souvisí s prvním, protože pokud bude granularita velká / malá, jediné váhy, které budou mít reálnou šanci se do batohu dostat, budou ty, co budou blízko maximu / minimu. Což povede k tomu, že veškeré vybrané váhy v batohu budou téměř stejné nebo úplně stejné. Tyto hodnoty se v obou případech ustálí na hodnotě $\frac{sumarniVaha}{pocetVeci}$. Zde se může vyskytnout námitka, že tento problém je způsobený dělením vah tak, aby splňovaly sumární váhu. To však není pravda, protože i v případě, že použijeme postup, kdy vygenerujeme náhodné váhy, sumární váhou bude jejich součet a kapacitu batohu nastavíme na násobek této hodnoty, budou poměrově váhy pořád stejné. Proto je dobré

nastavovat tento parametr v závislosti na sumární váze a na přiměřenou hodnotu.

Vygenerovaná instance má tento formát:

```
[počet věcí] [kapacita] [váha] [cena] [váha] [cena]...
```

Konkrétní případ:

```
6 20 8 32 7 2 8 25 4 41 6 34 7 26
```

Tento formát se používá v předmětu MI-PAA s tím rozdílem, že každý soubor obsahuje instance po řádcích, tudíž každá instance má na začátku index. Vzhledem k tomu, že generujeme instance po jednom, index chybí. Stále je však jednoduché vzít některou z instancí zadaných v MI-PAA a snadno ji nahrát do našeho systému.

3.3.2 SAT problém

Generátor problému splnitelnosti Boolovské formule obsahuje tyto parametry:

- počet proměnných,
- počet klauzulí,
- průměrný počet literálů v klauzuli.

Hlavním úkolem generátoru je vygenerovat daný počet klauzulí. Požadavkem je, aby byly klauzule generované náhodně, proto generování probíhá následovně:

1. Procházej postupně jednotlivé proměnné.
   a) S pravděpodobností
   $$\frac{1}{pocetPromennych/prumernyPocetLiteralu}$$
   rozhoduj o dané proměnné.
   b) Náhodně vygeneruj číslo 0, nebo 1.
   c) Pokud vygenerované číslo bylo jedna, přidej proměnnou do klauzule bez negace, pokud bylo mínus jedna, tak s negací.
2. Pokud instance ještě neobsahuje nově vygenerovanou klauzuli, přidej ji do instance. Pokud ano, pokračuj bez přidání.
3. Obsahuje-li instance požadovaný počet klauzulí, ukončí generování. Pokud ne, pokračuj krokem 1.

Krok, který říká, zda se bude o dané proměnné rozhodovat, je zde proto, že negenerujeme 3-SAT, tudíž počet proměnných v klauzuli může být libovolný. Průměrný počet literálů v klauzuli je jedním z parametrů generátoru.

Dalším důležitým krokem je kontrolování, zda je již nově vygenerovaná klauzule obsažena v instanci. Tuto věc řeším tak, že každou unikátní klauzuli použiji jako klíč do pole, ve kterém si uložím informaci, že tato klauzule je již v instanci. Výhodou tohoto řešení je, že kontrolu provedu v konstantním čase nezávisle na počtu generovaných klauzulí.

Jediným problémem mého řešení je generování takového počtu klauzulí, který se blíží maximálnímu možnému počtu klauzulí. Algoritmus totiž pracuje náhodně a poslední zbývající klauzule musí tedy „trefit“. Jediným řešením tohoto problému by bylo vygenerování všech možných klauzulí, ze kterých by se poté vybral zvolený počet. Velkým problémem tohoto řešení je však skutečnost, že takovýchto klauzulí je $3^n - 1$, kde n je počet proměnných. Tudíž by se dal použít pouze pro velmi malé instance, které nemá příliš cenu řešit iterativními metodami.

Vygenerovaná instance má DIMACS CNF formát:

```
p cnf [počet proměnných] [počet klauzulí]
[klauzule] 0
[klauzule] 0
...
```

Konkrétní případ:

```
p cnf 5 10
3 -5 0
-5 0
2 4 0
1 -2 -3 0
2 -3 0
-2 -3 0
-1 -2 0
1 2 0
-2 3 5 0
4 -5 0
```

3.3.3 Problém obchodního cestujícího

Generátor instancí problému obchodního cestujícího má tyto parametry:

- typ problému obchodního cestujícího,
- počet uzlů,
- počet hran,

- počet uzlů k navštívení,
- maximální cena hrany.

Jak jsem zmínil v kapitole 3.2.4, má implementace obchodního cestujícího obsahuje dva typy tohoto problému a od toho se také odvíjí generování instancí.

Pro moji variantu je jedinou podmínkou pro instanci problému to, že řešený graf musí být silně souvislý. Proto je třeba toto při generování zajistit, což řeším tak, že náhodně vyberu jeden z uzlů, z tohoto uzlu pak vytvořím hranu do jiného uzlu a tak dále. Počítaje s tím, že každý další uzel musí být jeden z nenavštívených, které si ukládám v poli. Každý vybraný uzel z pole odstraním, abych ho nemohl vybrat znovu. Tímto způsobem v $n - 1$ krocích vytvořím $n - 1$ hran, které graf spojí. Jeden z parametrů je také počet hran v grafu, proto je třeba náhodně vytvořit zbylé hrany. Důležité je také vědět, že všechny hrany v grafu jsou neorientované, proto je reálný počet vytvořených hran dvojnásobný. Na tuto věc je uživatel upozorněn v tooltipu tohoto parametru.

Pro typickou variantu obchodního cestujícího je třeba generovat úplný graf, čehož lze snadno dosáhnout.

Vygenerovaná instance má tento formát:

```
[typ]
[počet uzlů]
[počet hran]
[počet uzlů k~navštívení]
[maximální cena hrany]
*[vzestupně seřazený seznam uzlů k~navštívení, oddělený mezerami]
[graf jako 2D pole, odděleno mezerou, index uzel, hodnota hrana]
```

* Je obsažena pouze, je-li typ problému „Shortest“, což znamená varianta řešící reálnou nejkratší cestu.

Konkrétní případ:

```
Shortest
5
8
5
10
0 1 2 3 4
0 6 1 4 8
6 0 0 5 8
1 0 0 10 5
4 5 10 0 0
8 8 5 0 0
```

3.3.4 Problém minimálního uzlového pokrytí

Generátor instancí tohoto problému má tyto parametry:

- počet uzlů,
- počet hran.

Už z parametrů je jasné, že generování instancí pro tento problém je velmi jednoduché. Jediné, co generátor dělá, je, že do 2D pole, které popisuje graf, náhodně přidává jednotlivé hrany. Hrany nejsou orientované.

Vygenerovaná instance má tedy tento formát:

```
[počet uzlů]
[počet hran]
[graf jako 2D pole, odděleno mezerou, index uzel, hodnota hrana]
```

Konkrétní případ:

```
7
10
0 0 0 0 1 1 0
0 0 0 1 0 0 1
0 0 0 1 1 0 1
0 1 1 0 1 0 0
1 0 1 1 0 1 1
1 0 0 0 1 0 0
0 1 1 0 1 0 0
```

Kapitola 4

Uživatelské rozhraní

V této kapitole popíši změny provedené na uživatelském rozhraní oproti původnímu návrhu. Následně provedu heuristickou analýzu současného uživatelského rozhraní a v poslední řadě se podíváme na výsledky uživatelského testování.

Informace o implementaci najdete v práci Aplikace pro demonstraci funkce Tabu prohledávání [1]. Tuto práci vytvořil Jaroslav Veselý, který se zabýval implementací uživatelského rozhraní.

4.1 Změny oproti návrhu/prototypu

Oproti návrhu byly v současném uživatelském rozhraní provedeny některé změny, na které se nyní podíváme. Důvody pro tyto změny byly zjištěny během implementace nebo během uživatelského testování prototypu, které můžete najít v práci Aplikace pro demonstraci funkce genetických algoritmů [2].

Hlavními problémy prototypu byly tyto:

- Nemožnost určit parametry instancí.
- Generátor se nezavře po vygenerování.
- Lze zadávat nevalidní parametry.
- „Clear history“ maže historii napříč celým programem, nikoliv pouze u metody.

Všechny tyto problémy jsou v současné implementaci vyřešeny.

Popis implementace prototypu najdete v práci Aplikace pro demonstraci funkce Tabu prohledávání [1]. Jednalo se především o prototyp uživatelského rozhraní bez logiky.

4.1.1 Hlavní stránka

Současné uživatelské rozhraní najdete na obrázku 4.1. Z obrázku vidíme, že hlavní myšlenka návrhu zůstala stejná. Změnilo se pouze pár detailů jako:

- Tooltipy u parametrů a instancí.
  - Návrh neobsahoval tooltipy u parametrů a instancí. Bylo třeba je přidat, aby i nezkušený uživatel věděl, co jaký parametr znamená. U instancí obsahují tooltipy základní parametry instance.
- Odstranění položek z historie.
  - V návrhu chybělo tlačítko na odstranění položek z historie. V současném řešení je možné odstranit jak jednotlivé položky historie, tak smazat celou historii najednou.
- Prostřední část obrazovky.
  - Výraznější změnou prošel střed obrazovky. V původním návrhu se počítalo s tím, že tato část bude rozdělena na tři díly: graf, stavový prostor a výsledky. V současném řešení chybí stavový prostor, protože jsme dospěli k názoru, že pro větší instance problémů v něm není nic vidět, a proto nemá cenu ho vykreslovat. Další změnou je, že výsledky a graf byly spojeny do jednoho okna, kde graf je v horní části a výsledky se zobrazují pod ním. Výhodou tohoto řešení je, že uživatel nemusí nikam překlikávat a má veškerá data z průběhu na jednom místě.
- Informace po najetí na graf.
  - Další změnou, která nebyla uvedena v návrhu, je zobrazení informací po najetí na část v grafu. V současném řešení se po najetí zobrazí informace o daném kroku. V případě simulovaného ochlazování jsou těmito informacemi hodnota optimalizačního kritéria a teplota.
- Vícero instancí z panelu historie.
  - Další věcí, kterou můžeme na obrázku vidět, je zobrazení vícero výsledků v panelu historie. V takovémto případě se do grafu vykreslí oba průběhy a výsledky se zobrazí pod sebe. Při najetí na legendu v grafu si uživatel může tučně zobrazit pouze jeden z výsledků. Po kliknutí na tlačítko „+“ se zobrazí parametry, s jakými byla instance řešena, stejně jako je tomu v panelu historie.
- Nahrávání a generování instance.

  - Nahrávání a generování instance bylo v návrhu umístěno pod jedním tlačítkem. V současném řešení je tlačítko pro každou z těchto akcí.

- Nápověda, nastavení a podpora technologií v prohlížeči.

  - V pravém dolním rohu přibyla tlačítka s nápovědou, nastavením a tlačítko na kontrolu, zda daný prohlížeč podporuje používané technologie.

4.1.2 Okno generátoru

Okno generátoru viz obrázek 4.2. Jak můžeme z obrázku vidět, v případě generátoru je viditelná jediná změna, a tou je checkbox „Keep generator window open“. V původním návrhu bylo okno generátoru otevřené, dokud ho uživatel nezavřel. V současném řešení se zavře po generování, pokud není tato volba zaškrtnuta.

Další změnou je spíše změna funkcionality, kdy se název instance vytváří automaticky na základě zvolených parametrů.

Stejně jako v návrhu má u sebe každý parametr tooltip, kde je popsáno, co daný parametr znamená. Případně další doplňující informace, jako jsou například omezení.

4.2 Heuristická analýza

V této části projdu všechny body Nielsenovy heuristické analýzy, o které jsem již mluvil v části 2.6.1, a zjistím, jak moc je v současné době systém splňuje.

Viditelnost stavu systému

Jediné dvě části systému, kde probíhá delší výpočet a systém pracuje, jsou:

- řešení instance,
- generování instance.

Viditelnost systému je v těchto částech řešena změnou spouštěcích tlačítek na tlačítka ukončovací, což považuji za dostatečný indikátor toho, že systém právě na něčem pracuje. V případě řešení instance se také mění status vedle tlačítka „Start“.

Za menší problém by se dalo považovat načítaní grafů při změně metody. Pokud graf obsahuje spoustu hodnot je načítání delší, což by se mohlo zdát jako zamrznutí systému.

Shoda mezi systémem a realitou

Všechna tlačítka v systému jsou vybrána tak, aby symbolizovala svoji funkcionalitu. Jediný problém by mohl být u tlačítek na nahrání a generování instance. Z toho důvodu se po najetí myší na tato tlačítka zobrazí popis.

Minimální zodpovědnost

Kroky, které by se daly v systému považovat za nevratné, je smazání instancí a historie řešených instancí. Toto smazání lze dělat po jednom, kdy se po smazání zobrazí *toaster message*, ve které je uživateli umožněno vrátit smazání zpět. Druhou možností je smazat všechny instance nebo celou historii. V tomto případě je uživatel upozorněn, zda chce tuto akci opravdu provést, a musí ji potvrdit. Považuji tedy tento bod za splněný.

Shoda s použitou platformou a obecnými standardy

Systém obsahuje běžně na webu používané prvky. Popisky, které jsou v systému, opravdu dělají/znamenají to, co je na nich uvedeno.

Prevence chyb

Systém obsahuje validaci všech vstupních parametrů, a to jak v případě parametrů metod, tak generátoru. Uživatel je na problém upozorněn již při vyplnění a není mu umožněno danou akci provést, dokud nejsou parametry validní. Dále systém obsahuje kontrolu formátu nahrávaných souborů, kdy při nahrání nevalidního souboru není možné tento soubor spustit a je u něj uvedeno, jakou chybu obsahuje, aby ji mohl uživatel opravit.

Kouknu a vidím

Veškeré důležité informace jsou pro uživatele dostupné na právě zobrazené stránce. Zvolená metoda a instance je zvýrazněna, aby bylo jasné, která je aktuálně vybrána.

Flexibilita a efektivita

V systému je možné nastavit všechny potřebné parametry metody. Pro uživatele, který nemá s metodami zkušenosti, je vše ze začátku vyplněné, takže může rovnou stisknout tlačítko start. Metoda simulovaného ochlazování také obsahuje funkce na nastavení startovní teploty a velikosti vnitřního cyklu, což dále usnadňuje nezkušeným uživatelům práci.

Minimalita

Systém zobrazuje pouze všechny potřebné věci u dané metody. Stránka je strukturována tak, aby každá část obsahovala to, co je potřeba, a nic navíc.

Smysluplné chybové hlášky

Chybové hlášky jsou psané tak, aby uživateli poradily, kde je problém a jak ho má opravit.

Nápověda a dokumentace

Systém v dolním pravém rohu obsahuje nápovědu, ve které je vysvětleno vše, co by mohl uživatel potřebovat a co jsme ochotni mu prozradit. Vzhledem k tomu, že se jedná o výukovou aplikaci, nechceme studentům poskytnout detaily samotné implementace.

Shrnutí

Systém dle mého názoru splňuje všechny body Nielsenovy heuristické analýzy. Jedinou změnou by mohlo být přidání „loading znaku“ při výpočtu, generování instancí a vykreslování grafu při změně metody. Současné řešení však nepovažuji za nesplnění daného bodu.

4.3 Uživatelské testování

V rámci uživatelského testování jsme oslovili několik studentů, kteří absolvovali předmět MI-PAA, abychom zjistili jejich názor na naší aplikaci. Testování bylo rozděleno na několik částí:

1. otázky před testováním,
2. scénáře na testování UI,
3. scénáře specifické pro části, které jsme implementovali jako jednotlivci,
4. otázky na průběh testování.

Dále jsme také oslovili cvičící předmětu MI-PAA, aby si aplikaci prošli a poskytli nám zpětnou vazbu.

Videa z testování lze nalézt v příloze.

4.3.1 Položené otázky

Studentů jsme se ptali na tyto otázky:

- Otázky před testováním:

  - Jaký algoritmus jste si vybrali pro svou závěrečnou práci v předmětu MI-PAA?
  - Znáte i zbylé dva algoritmy?
  - Znáte applety, které sloužily k demonstraci běhu iterativních algoritmů?
  - Zúčastnili jste se testování prototypu naší aplikace?
  - Prohlédněte si aplikaci a sdělte nám, jak byste aplikaci začali používat.

- Otázky po testování:
  - Co se vám na aplikaci líbilo?
  - Co se vám nelíbilo nebo co vám přišlo složité/neintuitivní?
  - Co si o této aplikaci myslíte v porovnání s původními applety?
  - Myslíte si, že by vám tato aplikace v předmětu MI-PAA pomohla?

Odpovědi na otázky:

- Nguyen Thi Tuyet Trang
  1. Jaký algoritmus jste si vybrali pro svou závěrečnou práci v předmětu MI-PAA?
     - „Simulované ochlazování.“
  2. Znáte i zbylé dva algoritmy?
     - „Pouze jsem o nich slyšela.“
  3. Znáte bývalé applety, které sloužily k demonstraci běhu iterativních algoritmů?
     - „Neznám.“
  4. Zúčastnili jste se testování prototypu naší aplikace?
     - „Ano.“
  5. Prohlédněte si aplikaci a sdělte nám, jak byste aplikaci začali používat.
     - „Začala bych proklikáním záložek a poté bych spustila běh jednoho z nich.“
  6. Co se vám na aplikaci líbilo?
     - „Pěkné UI, grafová reprezentace.“
  7. Co se vám nelíbilo nebo co vám přišlo složité/neintuitivní?
     - „Instance se špatným formátem je těžké poznat. Stejně pojmenované instance v historii není možné rozeznat.“

8. Co si o této aplikaci myslíte v porovnání s původními applety?
   - „Původní applety jsem neviděla.“
9. Myslíte si, že by vám tato aplikace v předmětu MI-PAA pomohla?
   - „Spíše ano.“

- Marek Dostál

1. Jaký algoritmus jste si vybrali pro svou závěrečnou práci v předmětu MI-PAA?
   - „Simulované ochlazování.“
2. Znáte i zbylé dva algoritmy?
   - „Znám.“
3. Znáte bývalé applety, které sloužily k demonstraci běhu iterativních algoritmů?
   - „Znám.“
4. Zúčastnili jste se testování prototypu naší aplikace?
   - „Ano.“
5. Prohlédněte si aplikaci a sdělte nám, jak byste aplikaci začali používat.
   - „Klikl na *Start*.“
6. Co se vám na aplikaci líbilo?
   - „Vzhled, validace. Neseká se.“
7. Co se vám nelíbilo nebo co vám přišlo složité/neintuitivní?
   - „Nevěděl jsem, co znamenaly tabulky pod výsledky běhu.“
8. Co si o této aplikaci myslíte v porovnání s původními applety?
   - „Lepší, snadno spustitelné.“
9. Myslíte si, že by vám tato aplikace v předmětu MI-PAA pomohla?
   - „Spíše ano.“

- Petra Krnáčová

1. Jaký algoritmus jste si vybrali pro svou závěrečnou práci v předmětu MI-PAA?
   - „Simulované ochlazování.“
2. Znáte i zbylé dva algoritmy?
   - „Ano, ale už si je moc nepamatuji.“
3. Znáte bývalé applety, které sloužily k demonstraci běhu iterativních algoritmů?

   - „Ano.“
4. Zúčastnili jste se testování prototypu naší aplikace?
   - „Ne.“
5. Prohlédněte si aplikaci a sdělte nám, jak byste aplikaci začali používat.
   - „Prohlédla si tooltipy u jednotlivých parametrů a poté jednotlivé metody.“
6. Co se vám na aplikaci líbilo?
   - „Přehledné rozhraní. Spousta vysvětlivek, validace.“
7. Co se vám nelíbilo nebo co vám přišlo složité/neintuitivní?
   - „Ze začátku jsem neviděla historii, ale nemyslím si, že to je problém.“
8. Co si o této aplikaci myslíte v porovnání s původními applety?
   - „Přehlednější, novodobé a snadno spustitelné.“
9. Myslíte si, že by vám tato aplikace v předmětu MI-PAA pomohla?
   - „Ano, je to dobré na učení. Člověk si to může vyzkoušet, aniž by to programoval.“

- Dočekalů Danny

1. Jaký algoritmus jste si vybrali pro svou závěrečnou práci v předmětu MI-PAA?
   - „Simulované ochlazování.“
2. Znáte i zbylé dva algoritmy?
   - „Z toho, co jsem se učila na zkoušku.“
3. Znáte bývalé applety, které sloužily k demonstraci běhu iterativních algoritmů?
   - „Ano, ale šly mi špatně spustit.“
4. Zúčastnili jste se testování prototypu naší aplikace?
   - „Ne.“
5. Prohlédněte si aplikaci a sdělte nám, jak byste aplikaci začali používat.
   - „Proklikala metody a pak klikla na *Start*.“
6. Co se vám na aplikaci líbilo?
   - „Graf. Porovnání instancí.“
7. Co se vám nelíbilo nebo co vám přišlo složité/neintuitivní?

   - „Historie byla na začátku trochu matoucí, ale časem jsem ji používala bez problému.“

8. Co si o této aplikaci myslíte v porovnání s původními applety?
   - „Lehce spustitelné. Funkčnost. Lépe se s tím pracuje.“
9. Myslíte si, že by vám tato aplikace v předmětu MI-PAA pomohla?
   - „Ano, například při řešení domácích úloh.“

- Jan Nováček

1. Jaký algoritmus jste si vybrali pro svou závěrečnou práci v předmětu MI-PAA?
   - „Simulované ochlazování.“
2. Znáte i zbylé dva algoritmy?
   - „Ano.“
3. Znáte bývalé applety, které sloužily k demonstraci běhu iterativních algoritmů?
   - „Ne.“
4. Zúčastnili jste se testování prototypu naší aplikace?
   - „Ne.“
5. Prohlédněte si aplikaci a sdělte nám, jak byste aplikaci začali používat.
   - „Proklikal metody a pak klikl na *Start*.“
6. Co se vám na aplikaci líbilo?
   - „Rozložení. Interaktivní grafy.“
7. Co se vám nelíbilo nebo co vám přišlo složité/neintuitivní?
   - „Závorkování při stejných jménech. Popis u najetí na graf.“
8. Co si o této aplikaci myslíte v porovnání s původními applety?
   - „Applety jsem neviděl.“
9. Myslíte si, že by vám tato aplikace v předmětu MI-PAA pomohla?
   - „Například na ladění parametrů pro semestrální práci.“

- Jindřich Máca

1. Jaký algoritmus jste si vybrali pro svou závěrečnou práci v předmětu MI-PAA?
   - „Genetický algoritmus.“
2. Znáte i zbylé dva algoritmy?
   - „Ano, teoreticky.“

3. Znáte bývalé applety, které sloužily k demonstraci běhu iterativních algoritmů?
   – „Nevěděl jsem, že existují.“
4. Zúčastnili jste se testování prototypu naší aplikace?
   – „Ne.“
5. Prohlédněte si aplikaci a sdělte nám, jak byste aplikaci začali používat.
   – „Spustil bych všechny metody.“
6. Co se vám na aplikaci líbilo?
   – „Jednoduchost, přímočarost. Vše v jednom okně.“
7. Co se vám nelíbilo nebo co vám přišlo složité/neintuitivní?
   – „Problém pod parametry. Set u SA změnit na Auto.“
8. Co si o této aplikaci myslíte v porovnání s původními applety?
   – „Applety jsem neviděl.“
9. Myslíte si, že by vám tato aplikace v předmětu MI-PAA pomohla?
   – „Určitě.“

- Jakub Koza

1. Jaký algoritmus jste si vybrali pro svou závěrečnou práci v předmětu MI-PAA?
   – „Genetický algoritmus.“
2. Znáte i zbylé dva algoritmy?
   – „Ano z MI-PAA.“
3. Znáte bývalé applety, které sloužily k demonstraci běhu iterativních algoritmů?
   – „Ano.“
4. Zúčastnili jste se testování prototypu naší aplikace?
   – „Ne.“
5. Prohlédněte si aplikaci a sdělte nám, jak byste aplikaci začali používat.
   – „Proklikal bych si to a spustil.“
6. Co se vám na aplikaci líbilo?
   – „Dynamičnost.“
7. Co se vám nelíbilo nebo co vám přišlo složité/neintuitivní?
   – „Problém pod parametry. Tlačítko na generátor.“

8. Co si o této aplikaci myslíte v porovnání s původními applety?
   - „Funguje.“
9. Myslíte si, že by vám tato aplikace v předmětu MI-PAA pomohla?
   - „Ano.“

4.3.2 Testovací scénáře

Průběh testování studentů byl řízený pomocí testovacích scénářů, kterými jsme se snažili donutit studenta využít všechny funkcionality systému. Obecné scénáře byly tyto:

1. Jste znovu studentem předmětu MI-PAA, zkuste si spustit libovolnou metodu. Jako problém zvolte SAT.
   - Základní scénář, cílený na zvolení problému a spuštění běhu.
2. Zjistěte, s jakými parametry byl spuštěn první běh.
   - Tímto scénářem jsme chtěli uživatele přimět k použití historie.
3. Dle svého uvážení upravte parametry metody a spusťte řešení znovu.
   - Vyzkoušení si úpravy parametrů metody a příprava na další scénář.
4. Porovnejte výsledky obou běhů.
   - Opět použití historie, tentokrát však cílené na porovnání více běhů.
5. Přestalo Vás bavit spouštět pouze jednu instanci problému. Vložte do aplikace vlastní soubor, který naleznete na ploše. Soubor má jméno „spatna.cnf“.
   - Zde chceme zjistit, zda uživatel snadno najde tlačítko nahrání vlastního souboru. Zde také přišla otázka, zda by uživatel poznal, že je nahraný soubor nevalidní, pokud by se nejmenoval „spatna.cnf“.
6. Vypadá to, že v souboru je chyba. Zjistěte jaká a zkuste nahrát soubor se jménem „example.cnf“.
   - Doplňující otázka na rozpoznání nevalidního souboru. Zde jsme chtěli zjistit, jestli je jasné, že ikonka s červeným vykřičníkem obsahuje popis chyby v daném souboru.
7. Chcete další instanci SAT problému, vygenerujte si vlastní s libovolnými parametry.
   - Testování otevření a použití generátoru.

8. Zjistěte parametry nově vygenerované instance.

    - Scénář na použití popisku u instance, který obsahuje informace o parametrech dané instance.

9. Chcete vyzkoušet, co dál aplikace umí. Vygenerujte a následně spusťte instanci pro problém batohu.

    - Opět změna typu problému a také vygenerování instance.

10. Již se nevyznáte ve všech instancích a v historii řešení je také moc záznamů. Smažte všechny vstupní soubory a celou historii.

    - Testování použití tlačítka na smazaní historie a všech instancí.

Některé z mnou implementovaných částí byly již otestovány v předchozích testech. Pro otestování zbylých částí jsem použil tyto scénáře:

1. Zkuste vyřešit problém obchodního cestujícího metodou simulovaného ochlazování. Nastavte startovní teplotu na 40 a spusťte.

    - Otestování problému obchodního cestujícího a příprava na další scénáře.

2. Přijde Vám, že startovní teplota není ideální. Zkuste nastavit startovní teplotu automaticky. Minimální teplotu nastavte na 0,1.

    - Testování automatického nastavení startovní teploty. Minimální teplota snížena, aby se řešení ustálilo, což je důležité pro další scénář.

3. Spusťte řešení znovu a zjistěte, přibližně při jaké teplotě se řešení ustálilo a přestalo přijímat velká zhoršení.

    - Zde jsem se snažil zjistit, zda uživatel přijde na to, že při najetí na graf se zobrazuje kromě hodnoty optimalizačního kritéria také teplota v daném kroku.

4. Vygenerujte dvě různé instance se stejným jménem.

    - Zde jsem chtěl zjistit, zda je uživatel schopný rozpoznat stejnojmenné instance díky přidání číselné hodnoty v závorce, jako je tomu například v operačním systému Windows.

5. V generátoru u problému obchodního cestujícího zjistěte, jaký je rozdíl mezi generovanými typy.

    - Testování tooltipu v okně generátoru. Zároveň test, zda je uživateli jasné, jaký je rozdíl mezi implementovanými typy obchodního cestujícího.

4.3.3 Závěry z testování

Během testování se neobjevily žádně závažné chyby v uživatelském rozhraní nebo implementaci.

Mezi větší problémy by se daly zařadit tyto:

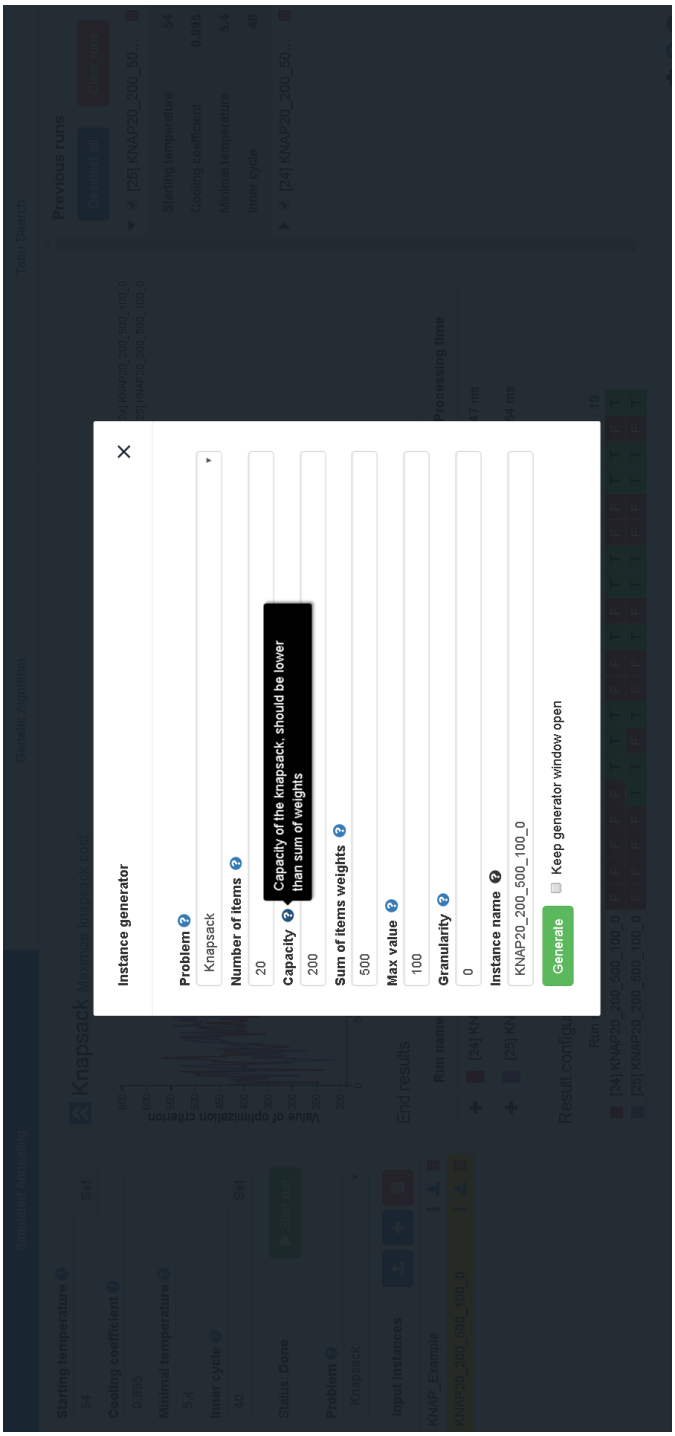
- Stejně pojmenované instance v historii není možné rozeznat.
  - Opraveno hned po testování.
- Umístění problému nad parametry metody.
  - Tento problém jsme v průběhu konzultovali s testery. Po vysvětlení důvodů pro toto umístění nám dali za pravdu a jedná se tedy spíše o preference jednotlivců než o problém.
- Těžko poznatelné nevalidní instance.
  - Tento problém při testování byl způsobem tím, že při testování byly v seznamu instancí pouze dvě instance: jedna vybraná a druhá nevalidní. V tomto případě je těžké nevalidní instanci poznat, protože označená instance má oranžovou barvu a nevalidní světle červenou. Při více instancích se nejednalo o problém.

Mezi menší problémy, které byly opraveny na základě testování, patří:

- upraveny popisky po najetí na graf,
- pojmenování y osy u simulovaného ochlazování a Tabu prohledávání,
- přidán popisek k výsledným konfiguracím zobrazeným po výpočtu.

Některé další připomínky nebyly shledány za velký problém, ale spíše za preference jednotlivců, proto nebyly změněny. Časově náročné návrhy nebyly přidány/změněny a jsou předmětem budoucích úprav této aplikace. Nejednalo se o problémy narušující funkcionalitu, ale spíše o „nice to have“ nápady a úpravy.

Obrázek 4.1: Hlavní obrazovka aplikace



Obrázek 4.2: Okno generátoru

Kapitola 5

Informace o systému

V této kapitole stručně shrnu doplňující informace o systému. Kdo se zabýval, kterými částmi systému a jak systém nasadit.

5.1 Shrnutí rozdělení systému podle diplomových prací

Zde bych rád krátce shrnul, v rámci jakých prací byly vytvořeny jednotlivé části systému, aby bylo jasné, kdo co vytvořil a v jaké práci najdete informace o jednotlivých částech systému / vývoje.

	Kluzáček M.	Kugler A.	Veselý J.
Návrh UI	✓	konzultant	konzultant
Implementace prototypu	konzultant	konzultant	✓
Testování prototypu	asistent	✓	asistent
Implementace společného UI	konzultant	konzultant	✓
Simulované ochlazování	✓		
Genetický algoritmus		✓	
Tabu prohledávání			✓
Graf průběhu		GA modifikace	✓
SAT	generátor		problém
TSP	✓		
Batoh	✓	problém	problém
MVC	✓		
ETSP		✓	
Závěrečné testování	SO	GA	Tabu
Nasazení aplikace			✓

Tabulka 5.1: Přehled rozdělení práce na aplikaci

Práce byla rovnocenně rozdělena na základě osobních preferencí a zkušeností s danými technologiemi, s výjimkou samotných metod, jejichž rozdělení bylo specifikované v zadání jednotlivých diplomových prací.

Rozdělení práce viz tabulka 5.1.

Části vytvořené Adamem Kuglerem najdete v práci Aplikace pro demonstraci funkce genetických algoritmů [2]. Práce Jaroslava Veselého má název Aplikace pro demonstraci funkce Tabu prohledávání [1].

5.2 Nasazení systému

Jediným požadavkem na prostředí, ve kterém lze aplikaci sestavit, je instalace softwaru *node.js*.

5.2.1 Postup nasazení ze souborů na DVD

Pro nasazení na server musíme v konfiguraci nastavit relativní nebo absolutní cestu k umístění souborů z hlediska url. Tato cesta je nastavena pomocí proměnné `build.assetsPublicPath` v konfiguračním souboru `/src/impl/config/index.js` na DVD. Například hodnota této proměnné pro testovací prostředí je: `'/dp-advanced-iterative-methods/'` pro následující url.

https://veselj43.github.io/dp-advanced-iterative-methods/

Kromě konfigurace už se jedná jen o sérii příkazů spuštěných v adresáři `impl`.

- Instalace (resp. stažení) závislostí `npm install` (lze spustit před úpravou konfigurace).
- Sestavení `npm run build`.
- Výsledné soubory jsou k nalezení v adresáři `impl/dist/`.
- Pro nasazení je zapotřebí zkopírovat obsah adresáře `impl/dist/` na server.

Jakmile jsou soubory sestavené s odpovídající konfigurací přístupné, aplikace je nasazena.

Postup nasazení je převzatý z práce Aplikace pro demonstraci funkce Tabu prohledávání [1].

Závěr

V rámci své práce jsem provedl analýzu systému na demonstraci běhu simulovaného ochlazování. Analýza se skládala z několika částí. V první části jsem provedl sběr požadavků, který se skládal z:

- uživatelského průzkumu, jehož cílem bylo získat informace od studentů předmětu MI-PAA a zjistit, co by rádi v aplikaci viděli,
- rešerše konkurenčních systémů, kde jsem se snažil zjistit, jestli existují podobné aplikace a jaké jsou jejich silné a slabé stránky.

Na základě získaných informací jsem poté sepsal funkční a nefunkční požadavky. Po sepsání požadavků bylo třeba zvolit technologie a navrhnout architekturu.

Dále jsem vytvořil části analytické dokumentace, ve kterých jsem se snažil vysvětlit základní funkcionalitu a logiku systému. Tato dokumentace se skládá z:

- diagramu případů užití,
- diagramů aktivit,
- doménového modelu,
- diagramu tříd.

V poslední částí analýzy jsem vytvořil návrh uživatelského rozhraní, u kterého jsem využil wireframů k vytvoření základního vzhledu aplikace.

V části realizace jsem se ve své práci zaměřil na implementaci logických částí systému. Přesněji na implementaci samotné metody simulovaného ochlazování, kde jsem implementoval také adaptační mechanizmus na výpočet startovní teploty.

Dále na implementaci problému obchodního cestujícího a problému minimálního uzlového pokrytí. Implementoval jsem generátory pro problémy:

- problém batohu,
- SAT,
- problém obchodního cestujícího,
- problém minimálního uzlového pokrytí.

Pro tyto problémy jsem také implementoval kontrolu formátu nahrávaných souborů.

Mezi další implementované části systému, které jsem v práci blíže nepopisoval, patří:

- konfigurace pro permutační problémy,
- okno generátoru,
- panel parametrů u simulovaného ochlazování.

U posledních dvou zmíněných jsem implementoval UI a kontrolu zadávaných parametrů.

Na závěr jsem provedl heuristickou analýzu výsledné aplikace a společně s kolegy, kteří pracovali na zbylých částech systému, jsme provedli uživatelské testování.

V poslední řadě jsem také vytvořil dokumentaci logických částí systému.

Literatura

[1] Veselý, J.: *Aplikace pro demonstraci funkce Tabu prohledávání.* Diplomová práce, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2018.

[2] Kugler, A.: *Aplikace pro demonstraci funkce genetických algoritmů.* Diplomová práce, České vysoké učení technické v Praze, Fakulta informačních technologií, Praha, 2018.

[3] Soltesz, D. L.: What Does JavaScript Do? *Techwalla [online]*, 2018, [cit. 2018-03-15]. Dostupné z: `https://www.techwalla.com/articles/what-does-javascript-do`

[4] HTML. *Computer Hope [online]*, 2017, [cit. 2018-03-15]. Dostupné z: `https://www.computerhope.com/jargon/h/html.htm`

[5] Duncan, R.: What is CSS? *A Simple Guide to HTML [online]*, [cit. 2018-03-15]. Dostupné z: `http://www.simplehtmlguide.com/whatiscss.php`

[6] How CSS works. *MDN web docs [online]*, 2018, [cit. 2018-03-15]. Dostupné z: `https://developer.mozilla.org/en-US/docs/Learn/CSS/Introduction_to_CSS/How_CSS_works`

[7] What is Vue.js? *Vue.js [online]*, [cit. 2018-03-15]. Dostupné z: `https://vuejs.org/v2/guide/`

[8] What is wireframing? *experienceux [online]*, [cit. 2018-03-17]. Dostupné z: `https://www.experienceux.co.uk/faqs/what-is-wireframing/`

[9] Simulated Annealing. *Carnegie Mellon University [online]*, [cit. 2018-03-23]. Dostupné z: `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/learn-43/lib/photoz/.g/web/glossary/anneal.html`

[10] A Comparison of Cooling Schedules for Simulated Annealing (Artificial Intelligence). *what-when-how[online]*, [cit. 2018-04-30]. Dostupné z: `http://what-when-how.com/artificial-intelligence/a-comparison-of-cooling-schedules-for-simulated-annealing-artificial-intelligence/`

[11] Ben-Ameur, W.: Computing the Initial Temperature of Simulated Annealing. *Computational Optimization and Applications*, ročník 29, č. 3, Dec 2004: s. 369–385, ISSN 1573-2894, doi:10.1023/B:COAP.0000044187.23143.bd. Dostupné z: `https://doi.org/10.1023/B:COAP.0000044187.23143.bd`

[12] Hordějčuk, V.: Obchodní cestující (Travelling Salesman). *voho[online]*, [cit. 2018-04-04]. Dostupné z: `http://voho.eu/wiki/optimalizace-tsp/`

Příloha **A**

Seznam použitých zkratek

UI User interface

MI-PAA Problémy a algoritmy

HTML HyperText Markup Language

CSS Cascading Style Sheets

SAT Boolean satisfiability problem

VLSI Very-large-scale integration

FPTAS Fully Polynomial Time Approximation Scheme

Příloha **B**

Obsah přiloženého DVD

- `readme.txt` ..... stručný popis obsahu DVD
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `DP_Kluzacek_Michal_2018.pdf` ..... text práce ve formátu PDF
- `documentation` ..... dokumentace
  - `analytic` ..... analytická dokumentace
  - `code` ..... dokumentace .js souborů
- `images` ..... obrázky
- `videos` ..... videa z uživatelského testování