ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Simulátor paralelních řadících algoritmů v prostředí webového prohlížeče
Student:	Bc. Petr Plechatý
Vedoucí:	Ing. Michal Šoch, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

Seznamte se s problematikou paralelních řadících algoritmů. Následně se seznamte s aktuálními technologiemi tvorby webových aplikací. Vyberte vhodnou technologii pro realizaci webové aplikace, která bude fungovat jako simulátor vybraných paralelních řadících algoritmů. Zaměřte se na grafickou podobu simulace, aby bylo zřejmé, jak vlastní řadící algoritmus funguje. Aplikaci poté navrhněte, prakticky naimplementujte, otestujte a řádně zdokumentujte.

Aplikace musí simulovat minimálně následující řadící algoritmy:
- sudo-liché řazení na 1-D mřížce,
- ShearSort na 2-D mřížce,
- bitonický MergeSort na hyperkrychli.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 16. listopadu 2016

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Diplomová práce

Simulátor paralelních řadících algoritmů v prostředí webového prohlížeče

Bc. Petr Plechatý

Vedoucí práce: Ing. Michal Šoch, Ph.D.

4. května 2017

Poděkování

Rád bych poděkoval vedoucímu práce Ing. Michalu Šochovi, Ph.D. za ochotu a odborné vedení. Dále bych rád poděkoval celé své rodině za podporu, kterou mi poskytovala po celou dobu mého studia.

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 4. května 2017 .....

Odkaz na tuto práci

Plechatý, Petr. *Simulátor paralelních řadících algoritmů v prostředí webového prohlížeče.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Práce se zabývá analýzou, návrhem a implementací aplikace pro webový prohlížeč, která simuluje běh některých paralelních řadících algoritmů. Implementace je napsána v Reactu a využívá moderní technologie pro vývoj single-page aplikací.

Klíčová slova React, simulace, HTML5, řadící algoritmy, funkcionální programování

Abstract

The thesis describes the analyse, design and implementation of an application running in a web browser. This application simulates some chosen parallel sorting algorithms. The implementation is written in React along with other modern technologies for development of single-page applications.

Keywords React, simulation, HTML5, sorting algorithms, functional programming

Obsah

Seznam obrázků

Úvod

Motivace

Řazení je jednou ze základních úloh, která se řeší v programování. Součástí mnoha algoritmů bývají právě řadicí algoritmy, které řeší seřazení vstupních dat podle specifikovaného pořadí. Z tohoto důvodu je dobré před studiem komplexnějších algoritmů nejprve dobře porozumět algoritmům řadícím.

Pro snadnější pochopení toho, jak samotný algoritmus pracuje, se dá využít simulátoru algoritmů. Takový simulátor je aplikace, která zobrazuje chod konkrétního algoritmu. Hlavním cílem takovéto aplikace je, aby uživatel co nejsnadněji danému algoritmu porozuměl, proto se současné simulátory zaměřují hlavně na grafické zobrazení algoritmu, ve kterém je jasně zobrazeno, jak jsou data reprezentována a jak se v jednotlivých krocích algoritmu s daty pracuje.

Simulátorů, které simulují chod sekvenčních řadicích algoritmů, existuje v součastnosti velká řada. S problémem sekvenčního řazení se setkal snad každý programátor. Když něco programujeme, tak dřív nebo později musíme nějaká data seřadit, zvláště, pokud hledáme efektivní řešení problému, tedy řešení v co nejkratším výpočetním čase. Z tohoto důvodu je poptávka po simulátorech, které by pomohly s pochopením těchto algoritmů, docela velká a proto také existuje spousta řešení.

Na poli paralelního programování je však situace odlišná. Paralelní programování využívá velký počet výpočetních jednotek k řešení daného problému. Většina aplikací v současnosti je však psána pro běžné osobní počítače, nebo mobilní zařízení, tedy víceméně sekvenčně a proto se programování pro celé sítě výpočetních jednotek nevěnuje taková pozornost. Bývá však součástí studií na technických univerzitách, zejména pak na ČVUT, kde je pochopení těchto algoritmů zapotřebí.

Cíl práce

Cílem práce je vytvořit aplikaci, která by simulovala paralelní řazení prvků. Důraz je kladen především na optickou vizualizaci dat, která by znázorňovala, jak vybraný algoritmus pracuje. Potřeba je zobrazit samotný graf celé sítě včetně všech přechodů mezi jednotlivými kroky algoritmu.

Součástí práce je analýza současných simulátorů a požadavků na aplikaci, následované druhou částí práce, která se zabývá návrhem uživatelského rozhraní a samotné aplikace. Ve třetí části se zabývám realizací aplikace od výběru technologií až po samotnou implementaci. V poslední části je popsáno testování výsledné aplikace.

Kapitola 1

Popis problému

Tato kapitola se věnuje rozboru problému řešeného v této práci. Na začátku popisuji teoretické pozadí celého problému, propojovací sítě, jejich strukturu a algoritmy, které s nimi pracují a které budou v rámci této práce simulovány.

Výpočetní jednotka

Výpočetní jednotku můžeme pro účely této práce vnímat jako samostatný procesor s vlastní lokální pamětí, který jako součást grafu propojovací sítě bude vyobrazen jako uzel tohoto grafu.

Propojovací síť

Základní komponentou, která hraje hlavní roli v problému paralelních algoritmů je propojovací síť. Na fyzické úrovni se jedná o síť nezávislých výpočetních jednotek, které mohou být vzájemně propojeny komunikačními kanály. Po těchto komunikačních linkách si mohou výpočetní uzly vzájemně posílat data. V abstraktní rovině se jedná o graf, kde každý uzel představuje jednu výpočetní jednotku a každá hrana, vedoucí mezi dvěma uzly, je komunikační kanál.

Sousední uzly

Každé dva uzly, které jsou propojeny hranou, označujeme jako sousední uzly. Tento pojem se bude v této práci poměrně často objevovat, neboť mezi sousedními uzly dochází k výměně dat, kterou je potřeba v simulaci zobrazit pomocí animace.

Paralelní algoritmus řazení

Cílem každého řadící algoritmu je transformovat vstupní neuspořádanou množinu prvků na uspořádanou.

Výchozí stav Na vstupu algoritmu bude neuspořádaná množina prvků, která bude rovnoměrně rozmístěna mezi výpočetní uzly grafu. Rovnoměrně je zde myšleno tak, že každý uzel bude mít ve své lokální paměti před spuštěním algoritmu stejné množství prvků.

Koncový stav Po skončení běhu algoritmu budou prvky na uzlech grafu seřazeny v pořadí, které je specifické pro každý algoritmus a síť. Tedy pokud bychom postupně vypsali prvky z uzlů v tomto pořadí, získali bychom setříděnou posloupnost prvků ze vstupní množiny.

Kapitola 2

Analýza

2.1 Analýza simulátorů

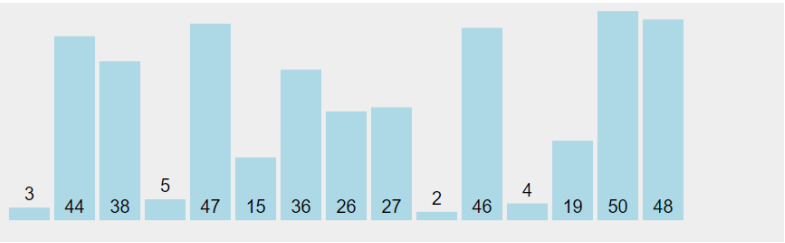
Jak už bylo zmíněno v úvodu, tak co se týká paralelních řadících algoritmů, nepodařilo se mi žádný veřejně dostupný nalézt. Naštěstí je v dnešní době na internetu spousta simulátorů, které simulují sekvenční algoritmy. Lze tedy při řešení této práce čerpat inspiraci alespoň u nich.

2.1.1 Visualgo.net

Ze všech možných řešení jsem k závěrečné analýze zvolil simulátor visualgo.net, který nejen že je veřejně dostupný na internetu, ale práce s ním je velmi intuitivní a grafické zpracování samotné simulace je jednoduché s dostatečným informativním charakterem.

Jak by možná napovídal název, dalo by se při analýze hledat v simulacích sekvenčních řadících algoritmů. V simulacích sekvenčních algoritmů bývají zobrazeny prvky (typicky čísla) v jedné řadě a následně se v každém jednotlivém kroku algoritmu tyto prvky přemisťují dokud nevytvoří seřazenou posloupnost prvků.

Jak je vidět na obrázku 2.1, můžou být vyobrazená čísla doplněna sloupci, kde výška každého sloupce odpovídá velikosti daného prvku v poměru k ostat-



Obrázek 2.1: Grafické pojetí řazení ve visualgo.net

Obrázek 2.2: Graf pro DFS ve visualgo.net

ním prvkům množiny. Toto zobrazení má tu nespornou výhodu, že pokud jsou vedle sebe zobrazeny dva sloupce, každý jiné velikosti, tak uživatel má informaci o tom, který prvek je větší zprostředkovánu mnohem rychleji, než když v hlavě porovnává dvě čísla.

Sekvenční řazení se bude v simulacích paralelních řadících algoritmů vyskytovat na jednotlivých uzlech, kdy bude potřeba po každém kroku, kdy procesoru přijdou data od sousedního uzlu, setřídit data lokálně.

Pro samotné vyobrazení celé komunikační sítě je ovšem potřeba hledat inspiraci jinde. Jelikož je síť komunikujících procesorů vlastně grafem, můžeme analýze podrobit grafové algoritmy. Konkrétně se jako velmi zajímavé jeví zobrazení algoritmů pro prohledávání grafu do hloubky (DFS - Depth-first search), kde je přehledně zobrazeno přecházení algoritmu z uzlu na jiný uzel grafu. Tuto část by bylo možné použít při zobrazení komunikace mezi procesory.

Zároveň nám simulátor visualgo.net nabízí i pohled na základní funkcionalitu, kterou bude obsahovat i výsledná aplikace, která je popisována v této práci. Jedná se konkrétně o tyto body:

Tlačítko spustit spustí běh animace, která se zastaví až s doběhnutím algoritmu, nebo při stisku tlačítka pro pauzu.

Pauza je tlačítko, které je dostupné pouze, když animace běží.

Krok vpřed je moc důležitá funkce, která umožňuje celou simulaci krokovat – spustí animaci, která se zastaví před dalším krokem algoritmu.

Krok zpět – podobně jako u předchozího tlačítka, akorát animace jde k předchozímu bodu. Toto je důležitá funkcionalita hlavně z toho důvodu, že se umožňuje vracet pouze o několik kroků, celou animaci spouštět znovu a tím umožňovat snadnější pochopení běhu samotného algoritmu.

Regulace rychlosti umožňuje simulaci zrychlovat nebo zpomalovat. To je dobré zejména, když se algoritmus nachází ve fázi, kterou uživatel zná

dobře a tím pádem ji může urychlit, nebo naopak si může zpomalit průběh animace u fáze, kterou teprve vstřebává.

Toto byl výčet fundamentální funkcionality, která byla i vybrána do požadavků pro aplikaci zpracované v této práci. Kromě toho je u grafu zobrazován i pseudokód s výrazněním řádku, na kterém se animace v daném okamžiku nachází. Tuto funkcionalitu by bylo dobré v rámci možností implementovat také – je to dobré pro lepší orientaci a při odhadování, za jak dlouho běh algoritmu skončí.

2.2 Požadavky

Před samotným rozborem požadavků na aplikaci je nutné vycházet z toho, jaký bude typický uživatel aplikace. Aplikaci zcela jistě nebudou používat lidé, kteří nevědí nic o programování nebo algoritmizaci, lze tedy předpokládat určité technické znalosti uživatele. Zároveň, pokud si uživatel aplikaci spustí, lze se domnívat, že se danou problematikou zabývá, nebo se alespoň nachází ve fázi, kdy se ji snaží pochopit. Z tohoto důvodu není potřeba v aplikaci vysvětlovat pozadí celé simulace, jako co jsou propojovací sítě, výpočetní jednotky a lze předpokládat, že když uživatel uvidí na obrazovce graficky znázorněnu síť, že ví, co která část znamená.

2.2.1 Funkční požadavky

Pro samotnou implementaci je důležité mít nadefinované požadavky na funkcionalitu aplikace, tedy co konkrétně bude aplikace umět. Jedná se o seznam všech důležitých funkcí systému. Některé z nich byly zmíněny již v rozboru existujících simulátorů 2.1.

1. Aplikace by měla vyobrazovat celou síť, se kterou daný algoritmus pracuje. Zobrazení by mělo být pomocí grafu, kde uzly grafů budou představovat výpočetní jednotky a hrany budou komunikační kanály, kterými si mohou uzly vyměňovat data.

2. Každý procesor může obsahovat 1 nebo více prvků ve své lokální paměti. Je tedy nutné myslet na to, že bude u každého uzlu grafu potřeba přehledně znázornit všechny jeho prvky.

3. Simulace musí jít spustit v režimu „play“, tedy bude plynule simulovat řazení až do stavu, kdy budou prvky seřazeny, potom simulace skončí.

4. Simulaci bude možné kdykoliv během průběhu animace zastavit a opětovně spustit od momentu posledního zastavení.

5. Simulací bude možno procházet v režimu „krok za krokem“. Budou od sebe odděleny jednotlivé kroky simulace, mezi kterými bude možno přecházet buď kliknutím na akci „krok vpřed“ nebo popřípadě „krok zpět“ pro navrácení simulace k předchozímu kroku algoritmu. Z tohoto vyplývá potřeba nadefinovat, co konkrétně znamená krok algoritmu. Krok algoritmu bude dvojího typu:

   - Paralelní krok – jedná se o výměnu prvků mezi sousedními uzly grafu, typicky bude vyobrazen jako tok prvků po hranách grafu.
   - Sekvenční lokální kroky – to jsou kroky, které probíhají uvnitř každého uzlu nezávisle na ostatních. Z celé animace by se tyto kroky daly vynechat, ovšem pro mnohem lepší přehlednost je lepší je zobrazovat. Tento bod nám definuje další důležitý funkční požadavek – zobrazování lokálního řazení na jednotlivých uzlech grafu.

   Celé „krokování“ simulace bude tedy umožněno jak na paralelní tak na sekvenční části algoritmu.

6. Musí jít nastavit velikost sítě, na které bude algoritmus probíhat.

7. Zároveň by měl jít nastavit počet prvků, které bude mít každý procesor v paměti. Spolu s předchozím bodem vnáší tento do celé simulace variabilitu, která napomůže lepší přehlednosti o tom, jak daný algoritmus pracuje.

8. Možnost volit si mezi implementovanými algoritmy - uživatel bude mít na výběr seznam všech dostupných algoritmů, mezi kterými bude moct přepínat.

2.2.2 Nefunkční požadavky

Definování nefunkčních požadavků vychází z hlavního požadavku na aplikaci, který říká, že se má aplikace zaměřit na grafickou podobu simulace, ze které je zřejmé, jak vlastní řadící algoritmus pracuje. Nejdůležitější je nastavit omezení aplikace tak, aby byla simulace přehledná. Nejdůležitějšími body jsou tyto:

1. Je dopředu nutné rozhodnou o podobě prvků, které budou v simulaci vystupovat. Simulace by mohla třídit například elementy s různým odstínem barvy - kdy by prvky řadila např. od nejsvětlejší po nejtmavší. Nebo by mohla řadit sloupce různé velikosti, jak bylo zmíněno v analýze simulátorů 2.1. Při zvážení typického příkladu naší simulace, kdy se jedná o graf s desítky uzlů, z nichž každý obsahuje několik prvků k seřazení, bude nejspíš nejvhodnější prvky reprezentovat čísly v rozsahu 0 až 99.

2. Zobrazovat pouze „rozumně velké“ grafy. Je nutné nastavit omezení, aby aplikace vykreslovala pouze grafy do určité velikost. Je potřeba počítat s tím, že graf by měl být během simulace vidět celý a aby žádná část simulace nebyla uživateli skryta. S tímto bodem souvisí i bod následující.

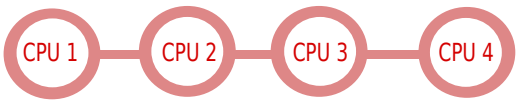
3. Určení mezních hodnot - mezní hodnoty se budou týkat počtu prvků, které budou obsahovat jednotlivé uzly grafu. Dalšími mezními hodnotami bude minimální a maximální rychlost přehrávání simulace.

4. Rozšiřitelnost - protože hlavním cílem aplikace je, aby srozumitelnou formou zobrazovala chod algoritmu a zároveň algoritmy nebudou počítat s velkými objemy dat (což by bylo právě v rozporu s přehledností), nebude při implementaci kladen důraz na efektivnost a rychlost. Tím pádem si lze dovolit směrovat aplikaci k větší rozšiřitelnosti. Nejzajímavější možnosti směrem k rozšiřitelnosti nabízí grafické pojetí aplikace.
   V ideálním případě by grafická stránka simulace měla být dokonale oddělena od logiky takovým způsobem, že pokud zaměníme grafické pojetí jakékoliv části simulace za zcela nové, nebudeme muset vůbec zasahovat do jiné části aplikace a zároveň by bylo možné mít např. řadu stylů, mezi kterými by bylo možné přepínat. Tímto způsobem by se dalo později obejít první omezení v tomto seznamu. Dopředu se ovšem nedá moc dobře rozhodnout, do jaké míry se v rozšiřitelnosti bude dát zajít. Hlavně co se týče otázky přidávání nových algoritmů do systému, protože každý algoritmus je hodně specifický a pracuje s jinou sítí. Ovšem je dobré si vytyčit alespoň základní vlastnosti, které by měly být dodrženy:

   - Přechody mezi jednotlivými kroky, které budou vyobrazeny jako animace, bude možné změnit bez zásahu do dílčí implementace jednotlivých algoritmů. Animace by tedy měli být absolutně nezávislé na jednotlivých grafech.
   - Grafické pojetí jednotlivých částí grafu (uzlů, hran) bude jednotné pro všechny algoritmy a bude možné je v budoucnu snadno změnit.
   - Možnost přidávání nových barevných schémat - do implementace by mělo být možné dodělat seznam barev, které změní barevné pojetí celé simulace.
   - Responzivita - u tohoto typu aplikace bylo počítáno s tím, že ji uživatel bude používat na počítači či laptopu. Aplikace nebude navržena pro malé displeje mobilních zařízení a podobně.

2.3 Analýza algoritmů

Pro analýzu dalších požadavků na aplikaci, je důležité se zaměřit na analýzu jednotlivých algoritmů, které by měla aplikace simulovat. Zajímají nás hlavně grafy, se kterými jednotlivé algoritmy pracují. Ostatní specifika algoritmů se



Obrázek 2.3: Graf 1-D mřížky pro Sudo-liché řazení

budou řešit až při samotné implementaci vnitřní logiky. Podle zadání bude naimplementována simulace následujících tří algoritmů:

- Sudo-liché řazení na 1-D mřížce
- ShearSort na 2-D mřížce
- bitonický MergeSort na hyperkrychli

Každý z těchto algoritmů má určitá specifika, která budou určovat omezení aplikace. U všech obrázků, na kterých je vyobrazena struktura grafu, jsou uzly označeny číslem, které udává pořadí, ve kterém jsou v konečné fázi algoritmu prvky seřazeny.

Sudo-liché řazení na 1-D mřížce

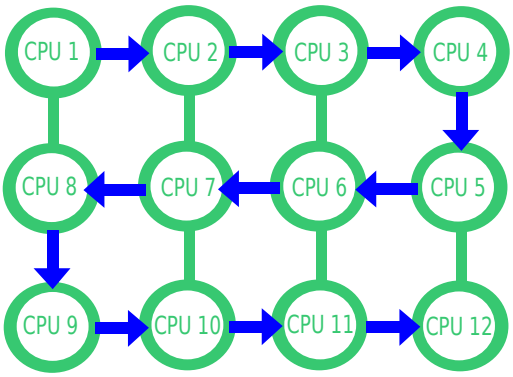
Struktura grafu pro tento algoritmus je jednoduchá - 1-D mřížka se skládá z několika procesorů, které jsou vzájemně propojeny hranami tak, aby vytvořily řadu (viz obrázek 2.3).

V každém paralelním kroku se vyměňují prvky mezi soudními uzly, které spojuje buď hrana vyskytující se v grafu jako lichá v pořadí nebo jako sudá. Tento graf se potom vyskytuje jako podgraf u následujícího algoritmu.

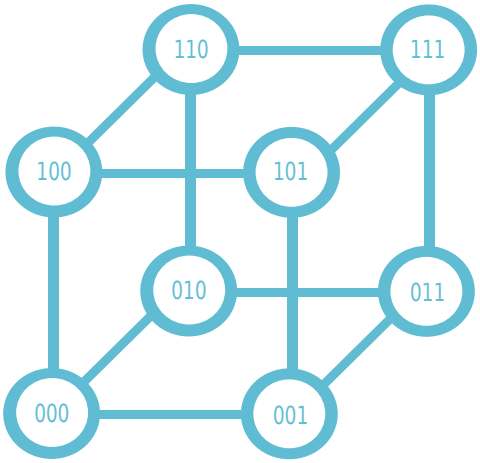
ShearSort na 2-D mřížce

2-D mřížka se skládá ze sloupců a řádku, přičemž je potřeba si všimnout hlavně toho, že Shearsort seřadí prvky na procesorech v řádcích střídavými směry, jako je vyobrazeno na obrázku 2.4.

Tento algoritmus v sobě obsahuje Sudo-liché řazení, jehož graf společně s grafem pro Shearsort lze poměrně snadno zobrazit. Sudo-liché řazení by se dalo označit za speciální případ Shearsortu na 2-D mřížce o jednom řádku.



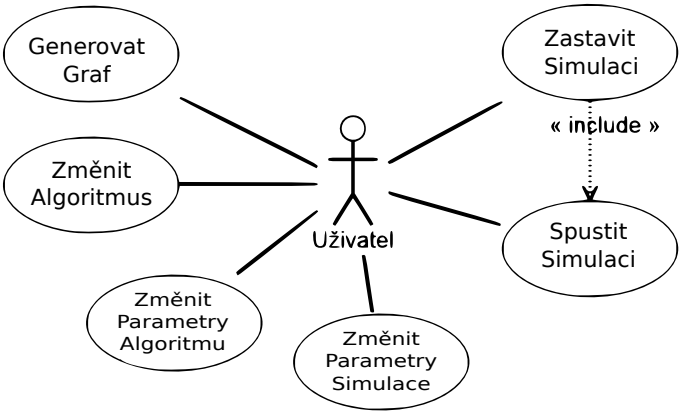
Obrázek 2.4: Graf 2-D mřížky pro Shearsort



Obrázek 2.5: Graf 3-D hyperkrychle

MergeSort na hyperkrychli

Zcela odlišná situace ovšem nastává u hyperkrychle. 2-D krychle je v podstatě mřížka jako u předchozího algoritmu, ale s rostoucím počtem dimenzí nastává problém, jak vlastně celý graf zobrazit. Vezmeme-li dvě 3-D krychle (viz obrázek 2.5) a zobrazíme je vedle sebe, přičemž doplníme chybějící hrany, dostaneme 4-D krychli. Pokud tento postup zopakujeme, dostaneme 5-D krychli atd. Takhle bychom mohli teoreticky zobrazit mnohorozměrné hyperkrychle, ovšem narazíme na problém, že u vyšších počtů dimenzí již bude graf značně nepřehledný. Z analýzy tohoto algoritmu vyplynulo stanovení horní meze pro počet dimenzí, které bude simulátor zobrazovat.



Obrázek 2.6: Případy užití

Shrnutí

Během analýzy algoritmů jsme si ujasnili hlavně jakou podobu budou mít grafy, na kterých algoritmy běží. Což je dobré vědět hlavně pro budoucí návrh grafického rozhraní. Zjistilo se, že se grafy zobrazují naprosto rozdílným způsobem - na to se bude muset myslet při budoucí implementaci.

2.4 Případy užití

Uživatelská role je ve webové aplikaci pouze jedna. Nepředpokládáme, že bychom chtěli některé algoritmy skrývat před určitou skupinou lidí a tak budou všechny funkce aplikace dostupné každému. Nemusíme řešit žádné přihlašování a podobně.

Na následujících stránkách v této sekci jsou popsány případy užití simulátoru z pohledu uživatele (viz obrázek 2.6) včetně stručných popisů, co se bude dít uvnitř aplikace v reakci na akce vyvolané uživatelem.

2.4.1 Změnit algoritmus

Po spuštění aplikace je toto jediná dostupná funkce. Potom, co si uživatel zvolí algoritmus, budou již dostupné i všechny ostatní akce.

- Uživatel klikne na nabídku „vybrat algoritmus“. Zobrazí se mu seznam všech dostupných algoritmů.

- Potom, co si z nabídky nějaký algoritmus vybere, rovnou se vygeneruje graf s předem definovanými *defaultními* rozměry. Zároveň budou rovnou

vygenerovány náhodné hodnoty a předpočítána simulace. Takže bude simulace rovnou připravena ke spuštění.

2.4.2 Generovat graf

- Simulace je již zobrazena. Uživatel klikne na možnost „Vygenerovat graf“. Dojde k zastavení doposud běžící simulace (pokud nějaká běží).
- Vezmou se nastavené parametry jako počet hodnot na procesor a velikost grafu a dojde k vygenerování nového grafu.
- Spočítá se nová simulace a uživateli se zobrazí vygenerovaný graf ve výchozím bodu simulace.

2.4.3 Změnit parametry algoritmu

Parametry algoritmu jsou následující:

- Velikost grafu - různé možnosti pro každý algoritmus
- Počet prvků na jeden uzel grafu

Při změně kteréhokoliv parametru dojde k vygenerování simulace s novými parametry a překreslení celé scény. Simulace se navrátí do počátečního stavu. Změna parametrů bude probíhat tímto způsobem:

- Uživatel u indikátoru počtu prvků na uzel grafu klikne na tlačítko „přidat“ nebo „odebrat“.
- Následně se počet prvků zvýší (nebo sníží) přesně o jeden.
- Pokud hodnota dosáhne mezního stavu, bude tlačítko s touto akcí deaktivováno.

  Podobný scénář bude i u všech ostatních parametrů. Možná výjimka bude, pokud nastavovaných hodnot bude málo (třeba počet dimenzí u hyperkrychle). V tomto případě bude akce vyobrazena jako nabídka možností.

2.4.4 Změnit parametry simulace

Tyto parametry se od těch předchozích liší tím, že po změně negenerují nový graf (dojde pouze k překreslení s novými parametry). Z tohoto důvodu se ve výčtu těchto parametrů objevují i ty, které přímo nesouvisí se samotnou simulací.

Změna rychlosti přehrávání

Uživatel bude mít k dispozici dvě tlačítka na změnu rychlosti přehrávání a číslo udávající aktuální rychlost. Toto číslo bude v rozsahu 1 (nejpomalejší) až 10 (nejrychlejší).

- Uživatel klikne na tlačítko pro zvýšení nebo snížení rychlosti.
- Rychlost animace se sníží (nebo zvýší) přesně o jeden stupeň.
- Pokud právě probíhá přehrávání simulace, aplikace na změnu rychlosti okmažitě zareaguje.

Změna grafického stylu

- Uživatel klikne na tlačítko pro změnu vzhledu.
- Zobrazí se nabídka s názvy dostupných stylů.
- Po zvolení stylu dojde k překreslení celého grafu. Přičemž stav simulace zůstane zachován. K této akci může dojít i během režimu přehrávání.

2.4.5 Spustit simulaci

Tlačítko pro spouštění simulace je dostupné pouze pokud již simulace neběží nebo pokud již nedoběhla. Pokud simulace běží, je toto tlačítko nahrazeno tlačítkem pro zastavení simulace.

1. Uživatel klikne na tlačítko „spustit“.
2. Pokud byla aplikace zastavena, spustí se přehrávání od okamžiku zastavení. V opačném případě se pokračuje krokem číslo 3.
3. Simulace si nahraje další předpočitaný stav a spočítá k němu přechody (animace) od aktuálního stavu.
4. Po skončení přehrávání přechodů se opakuje krok 3 až do ukončení celé simulace.

2.4.6 Zastavit simulaci

Podobně jako v předchozím případě je tato možnost dostupná pouze v případě, že simulace již běží a tlačítko umožňující tuto akci bude nahrazovat tlačítko pro spuštění simulace.

- Uživatel klikne na tlačítko „pauza“.
- Simulace se zastaví. Přičemž musí být zobrazen přesně stav, ve kterém se simulace nacházela v době zastavení.

2.4.7 Ostatní akce

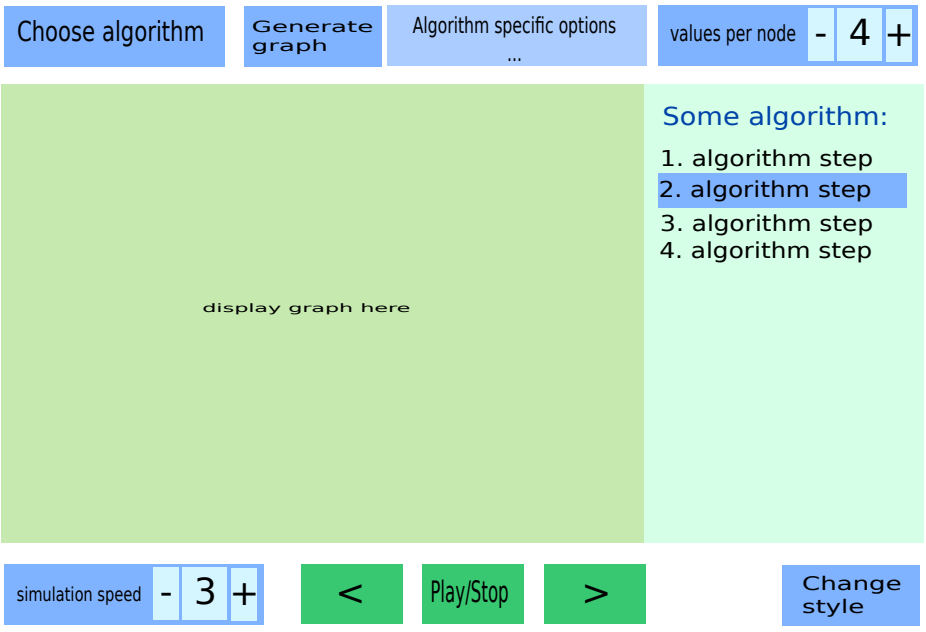
Toto ovšem nejsou všechny akce, které jsou uživateli k dispozici. Akce pro „krokování“ simulace mají v podstatě stejný scénář jako spouštění simulace 2.4.5. Rozdílem je, že po skončení aktuálního kroku se již nenačítá další a při akci „krok zpět“ se načítá předchozí stav namísto následujícího.

Kapitola 3

Návrh

3.1 Uživatelské rozhraní

Prvním krokem návrhu bylo si grafiky zobrazit základní elementy GUI, které se budou v aplikaci vyskytovat. Tyto prvky by měly obsáhnout všechny požadavky 2.2, které byly stanoveny během analýzy. Smyslem bylo vytvořit jakýsi prototyp, neboli mock-up, od kterého by se mohl odvíjet další návrh uživatelského rozhraní.



Obrázek 3.1: Schéma GUI

Obrázek 3.2: Základní CSS šablona

Výsledek je vidět na obrázku 3.1. Kromě toho, že obsahuje všechny funkce aplikace, zobrazuje rovněž základní rozmístění jednotlivých funkcí v aplikaci. Základní rozmístění funkcí je:

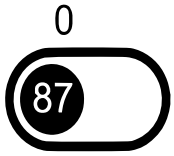
- Tlačítka pro ovládání simulace (na obrázku 3.1 zelená tlačítka), která se starají o funkci přehrávání/zastavení simulace, kroku vpřed a kroku zpět, jsou pohromadě situovány do středu dolní části aplikace. Zároveň by se poblíž těchto tlačítek měla nacházet funkcionalita pro ovládání rychlosti simulace. Smyslem je umístit všechna tlačítka, jejichž funkce spolu nějak souvisí, na jedno místo.

- Prostor pro zobrazení grafu by měl být co největší, proto je umístěn do středu. Je však omezený sloupcem, který vypisuje jednotlivé kroky algoritmu.

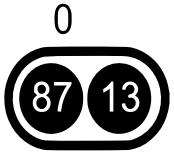
- Všechna nastavení, která neovlivňují chod simulace, ale mění pouze parametry grafu nebo algoritmu, jsou umístěna v horní liště. Každý algoritmus má ovšem jiné vstupní parametry, tohle bude potřeba při návrhu také zohlednit.

3.1.1 CSS šablona

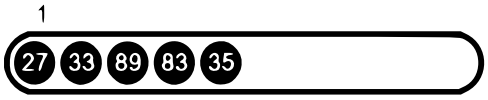
Potom, co bylo rozhodnuto o základním rozložení prvků, byla vytvořena šablona stylů. Pro nastylování byly použity standardní kaskádové styly CSS. Jelikož se, co se týká GUI, nejedná o nikterak složité rozhraní, nebyla použita žádná předpřipravená šablona nebo Bootstrap. Jak je vidět na obrázku 3.2, všechna tlačítka, lišty a nápisy jsou pojaty minimalisticky aby byl vytvořen prostor pro graf, který byl navržen později.



Obrázek 3.3: Uzel grafu s jedním prvkem



Obrázek 3.4: Uzel grafu s prvkem od sousedního uzlu



Obrázek 3.5: Uzel grafu s pěti prvky

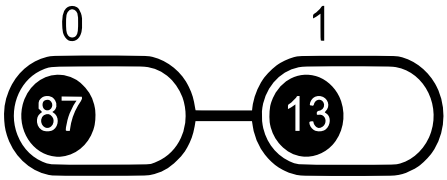
3.1.2 Grafické pojetí simulace

Nejdůležitějším grafickým návrhem je návrh podoby, jakou bude mít samotná simulace. Základním prvkem je uzel zobrazovaného grafu, který musí zabírat co nejmenší prostor a zároveň musí zobrazovat seznam všech prvků, které se v něm nachází spolu s číslem udávající jeho pořadí v grafu.

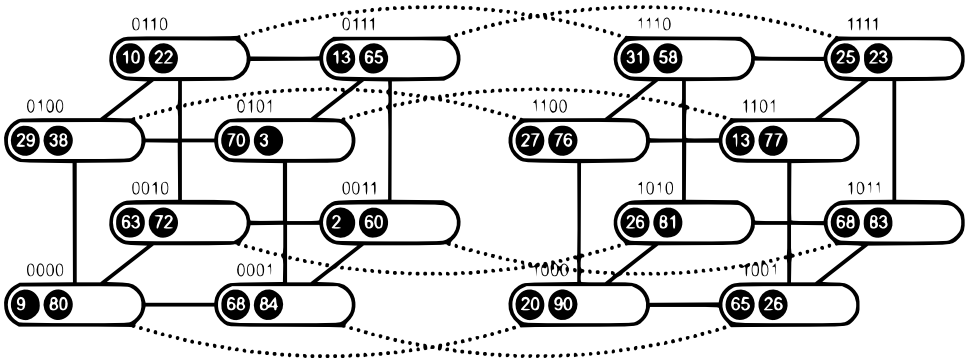
Uzel

Samotný uzel s pouze jedním prvkem je vyobrazen na obrázku 3.3. Jak je vidět, je potřeba dopředu počítat s tím, že počet prvků v uzlu se může zdvojnásobit, proto obsahuje extra místo pro další prvek. Ke zdvojnásobení počtu prvků bude docházet v situaci, kdy do uzlu dorazí prvky od sousedního procesoru. Taková situace je zobrazena na obrázku 3.4.

V porovnání s dalším obrázkem 3.5 je vidět, jak roste velikost uzlu v závislosti na počtu prvků na začátku simulace. Z tohoto důvodu byl omezen maximální možný počet prvků na každý procesor na 10.



Obrázek 3.6: Sousední uzly



Obrázek 3.7: 4-D hyperkrychle

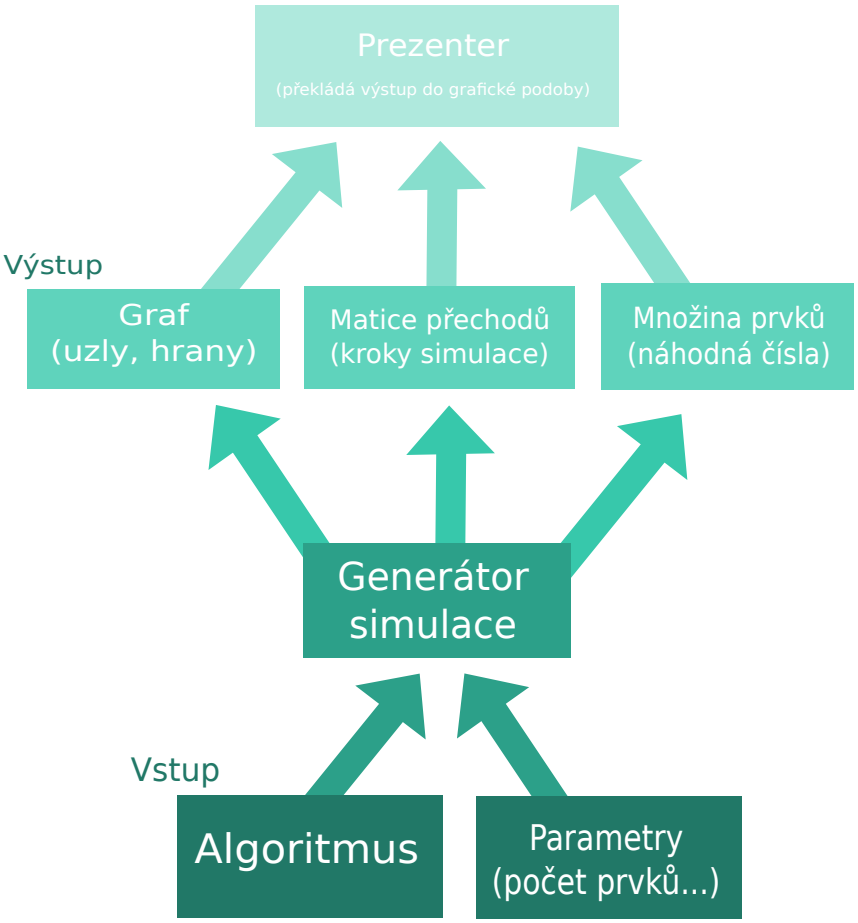
Hrany

Sousední uzly se již pouze propojí hranou (viz obrázek 3.6). Co se týká hran, tak zajímavější situace nastává v případě hyperkrychle, která ve vyšších dimenzích začíná být právě kvůli hranám nepřehledná. Z tohoto důvodu bude potřeba pro dimenze 4 a 5 zobrazovat hrany podobně, jako je tomu na obrázku 3.7. (Pro vyšší dimenze již graf zobrazován nebude - viz požadavky na aplikaci 2.2.)

3.2 Jádro aplikace

Jádrem aplikace je myšlena celá vnitřní logika, tedy jak bude aplikace pracovat. Na začátku celého procesu stojí uživatel, který definuje vstup celé simulace. Vstupem je algoritmus, parametry algoritmu (jako je např. počet řádků a sloupců u 2-D mřížky) a obecné parametry jako je počet prvků na každý uzel grafu.

Tento vstup je dále předložen generátoru simulace, který z předložených vstupních podmínek a naimplementované vnitřní logiky vytvoří výstupní data výsledné simulace. Výstupní data jsou trojího typu:



Obrázek 3.8: Schéma simulace

1. Vygenerovaná množina prvků – počet prvků je dán vstupem. Každý prvek bude obsahovat *id* a hodnotu, což bude náhodně vygenerované číslo z předem daného intervalu.

2. Vygenerovaný graf obsahující počáteční nastavení simulace. Každý uzel grafu bude obsahovat své souřadnice, takže generátor musí tyto souřadnice sestavit, aby výsledný graf měl svoji podobu. Zároveň bude uzel obsahovat seznam *id* prvků, které se nachází v jeho počáteční konfiguraci.

3. Matice přechodů simulace – obsahuje seznam stavů, kterými si simulace musí projít až do koncového stavu.

Tato data jsou předložena prezentační vrstvě aplikace, která je správně umístí a zobrazí do dokumentu webové stránky. Celý tento proces je zobrazen na schématu 3.8.

3.3 Technologie

Celá aplikace poběží čistě ve webovém prohlížeči. Odpadá tedy nutnost napojování aplikace do databáze, nebo komunikace s aplikacemi třetích stran. Tento fakt hrál důležitou roli při volbě technologií. Dalším důležitým faktem je, že pro implementaci není tolik důležitá efektivita jako přehlednost a znovupoužitelnost kódu, jak bylo popsáno v kapitole 2.1.

V dnešní době se hodně mluví o tzv. *single-page* aplikacích [1] a funkcionálním programování [2]. Tento přístup je jako dělaný pro aplikace tohoto typu. Celá aplikace bude mít na serveru podobu HTML stránky s importovanými styly a *javascriptovým* souborem obsahujícím celou aplikační logiku. Po načtení stránky do paměti prohlížeče již celá aplikace poběží v samotném prohlížeči bez nutnosti dotazovat se na server – toto zajistí implementace aplikace pomocí *single-page* přístupu. Další výhodou bude striktní oddělení dat od jejich prezentace – ve výsledku je možné překreslovat jenom části *UI* bez nutnosti překreslovat celou stránku a zároveň bude zajištěno, že pokud se data změní, bude zobrazena ve všech částech aplikace jejich aktuální podoba.
(O tohle vše se postarají knihovny, které jsou popsány dále.)

3.3.1 Nástroje pro vývoj

Pro celou aplikaci byl zvolen standard **ECMAScript 2015**, což je moderní nástupce klasického JavaScriptu. [3] Tento standard ovšem není zcela podporován webovými prohlížeči a tak je potřeba kód nejprve zkompilovat do „původního“ JavaScriptu. K tomu slouží nástroj *Babel*. Pro samotný vývoj byly pak použity nástroje jako *gulp* a *webpack*, které usnadňují vývoj především tím, že po každé změně kódu jej zkompilují a aktualizují výstup v prohlížeči. Tím pádem pokud například změníme vzhled nějaké komponenty,

projeví se tato změna téměř okamžitě bez nutnosti aktualizovat stránku webového prohlížeče. *Webpack* zároveň vytváří minifikované styly a skript obsahující aplikaci. Tím je potom zaručeno, že např. styly, které se v aplikaci nikde nepoužívají, ale přesto jsou ve zdrojových kódech, nebudou ve výsledné aplikaci zahrnuty.

3.3.2 Ramda a funkcionální přístup

Mezi základní paradigmata funkcionálního programování patří:

- funkce vyššího řádu (High-Order Functions),
- uzávěr (Closure),
- čistě funkcionální funkce (Pure Functions),
- neměnnost stavu (Immutability).

Dodržování základních zásad funkcionálního programování má spoustu výhod obzvláště pro samotný vývoj, udržovatelnost a znovupoužitelnost kódu. Nejzajímavější z pohledu naší aplikace jsou z výše uvedeného seznamu poslední dvě – čisté funkce a neměnnost stavu.

Neměnnost stavu

V imperativním programování dochází často ke změně vnitřního stavu, který je například při použití objektově orientovaného programování reprezentován vnitřním stavem každého objektu. Naproti tomu funkcionální přístup neumožňuje stav měnit - namísto toho je při každé změně vytvořena kopie celého stavu, která již v sobě tuto změnu zahrnuje. Všechny předešlé stavy jsou i nadále uloženy a to nám umožňuje například vracet se v čase. Pokud tedy např. voláme dvě funkce na jedno a to samé pole, kde každá toto pole modifikuje, vrátí každá funkce pole nové a zároveň si každá může být jistá, že pracuje s původními daty a nikoliv s již změněnými. Na rozdíl od stavů, které mohou být měněny, přináší ty neměnné méně vedlejších efektů (méně chyb) a jsou snadnější pro pochopení [4].

Čisté funkce

Pojem čistá funkce navazuje na neměnnost stavu tím, že každá funkce je chápána přesně tak, jak je chápána v klasické matematice. Funkce dostává data na vstup a na výstup pak vrátí nová data. Neexistuje zde žádný vnitřní stav, který by mohl výstup funkce změnit. Je tedy zaručeno, že pro jeden vstup funkce vrátí vždy stejný výstup. Nemůže tedy nikdy pro dva stejné vstupy vrátit různé výstupy. Dodržování tohoto paradigmatu bude kladen velký důraz při implementaci, aby se zabránilo zanášení nevyžádaných vedlejších efektů do aplikace.

Ramda

Ramda je knihovna napsaná pro *JavaScript*, jejíž cílem je proměnit jej v plně funkcionální jazyk [5]. Knihovna nabízí soubor funkcí pro manipulaci s objekty, daty, poli a jinými funkcemi. Všechny splňují paradigmata funkcionálního programování 3.3.2. Používání této knihovny tedy usnadňuje dodržovat pravidla funkcionálního programování.

Ve výsledku nám pak z kódu úplně zmizí proměnné, místo nich se budou v kódu vyskytovat konstanty. Pokud bude potřeba pracovat s upravenými daty, tak nad nimi zavoláme funkci, jejíž výstup uložíme do nové konstanty.

Existují i jiné knihovny, jejichž cílem je rovněž posunout JavaScript směrem k plně funkcionálnímu paradigmatu, jako například *Lodash*. Nicméně největší výhodou Ramdy je, že na první pohled vůbec nepracuje s daty. Většina kódu pracuje pouze s funkcemi, jejichž vzájemnou kompozicí vzniká celá logika. Toto umožňuje tzv. *currying*, což by se dalo přeložit jako „možnost dále rozvíjet funkci“. Ve skutečnosti se jedná o fakt, že funkci můžeme volat i s menším počtem parametrů, než kolik jich na vstupu potřebuje. V takovém případě tato funkce vrátí další funkci, která na vstupu bere parametry zbývající. Toto se opakuje až do chvíle, kdy je funkci předán poslední parametr (při programování ten nejvíc vpravo). Tento koncept umožňuje do proměnných ukládat funkce zavolané již s některými parametry, aniž bychom potřebovali funkci předkládat data. Této skutečnosti využívá Ramda, kde všechny funkce jsou naimplementovány tak, že se s daty pracuje až na konec. Jedná se tedy vždy o poslední parametr každé funkce[6].

Zároveň máme k dispozici funkci *Ramda.curry*, která umožňuje z jakékoliv funkce, která má na vstupu více než jeden parametr, udělat funkci s těmito vlastnostmi.

3.3.3 React

Stromovou reprezentací HTML stránky je *DOM (Document Object Model)*. Na DOM se dá v podstatě nahlížet jako na globální strukturu, do které lze ukládat stav naší aplikace. DOM však bývá velmi velký a práce s ním je opravdu pomalá [7]. Proto byl společností Facebook vytvořen *React*. V Reactu si na základě vstupu definujeme, jak má výsledná stránka vypadat. React ji potom sestaví do svého virtuálního DOMu, který následně porovnává s původním DOMem a aktualizuje pouze ty komponenty, které se liší. Jak vypadá taková komponenta zapsána v Reactu, je vidět v ukázce kódu 3.1.

```
1 import React from 'react';
2 import { map } from 'ramda';
3 import SimpleButton from './SimpleButton';
4
5 const ChangeAlgorithmButton = ({ buttons }) => (
6   <li>
```

```
7       <i className="icon-calculator3"/>
8       <ul>
9       {
10        map(button =>
11        <SimpleButton> {button.title} </SimpleButton>,
12        buttons
13        )
14      }
15      </ul>
16    </li>
17  );
18
19  export default ChangeAlgorithmButton;
```

Kód 3.1: Ukázka React komponenty

Na první pohled se zdá, že React míchá JavaScript s HTML. Ve skutečnosti tomu tak ale vůbec není, protože všechny značky, které jsou v kódu použity, jsou ve skutečnosti funkcemi. Takže výsledná komponenta je vlastně pouhou kompozicí funkcí. Tento zápis má ovšem obrovskou výhodu pro programátora, který si dokáže okamžitě představit, jak bude komponenta reprezentována v HTML kódu. Každá komponenta v Reactu je třída dědící z *React.Component* jejíž jedinou povinnou metodou je metoda *render()*, která právě definuje její podobu v DOMu. Pokud si naše komponenta vystačí pouze s touto vykreslovací metodou, lze celý zápis ještě zjednodušit a napsat celou komponentu ve tvaru funkce (jak je vidět v ukázce 3.1). Aplikace by se měla skládat z mnoha malých komponent, které by měly být zapsány v krátké podobě, raději než z velkých komponent ve kterých by později bylo snadné se ztratit.

3.3.4 Redux

Pro snadnější manipulaci s daty a jejich distribuci do všech potřebných komponent byl vyvinut *Redux* [8]. Ten se skládá z těchto hlavních částí:

- **Store** - jedná se o jakýsi globální kontejner, který v sobě obsahuje stav aplikace. Data jsou v tomto případě úplně oddělena od komponent aplikace. Ovšem je potřeba zařídit, aby se každá komponenta překreslila pokaždé, kdy dojde ke změně dat, se kterými daná komponenta pracuje. K tomu slouží v Reduxu metoda *subscribe*, kterou použijí komponenty napsané v Reactu k tomu, aby byly o případných změnách informovány.
- **Akce** oznamují, že došlo ke změně stavu aplikace. Akce jsou vyvolány metodou *Store.dispatch*.
- **Reducer** je funkce, která je použita ve *Storu* a která je zodpovědná za změnu stavu. Každý *reducer* musí být čistá funkce (viz 3.3.2).

Z pohledu programátora je potom hlavně potřeba nadefinovat akce a naprogramovat *reducery*. Každý *reducer* by se měl starat o ty části stavů aplikace, které spolu souvisí. Například jeden se bude starat o stavy uživatelského rozhraní, další pak bude měnit stav grafu atd.

Jedinou možností, jak změnit stav kterékoliv části aplikace, je vyvolat akci, na kterou zareagují příslušné reducery. Redux se potom sám postará o informování změn u všech komponent, kterých se změna týká.

Single source of truth

Redux je navžen tak, aby architektura aplikace dodržovala podmínky toho, čemu se v informatice říká *Single source of truth*. Tento pojem vlastně říká, že data jsou v aplikaci uložena pouze na jednom místě a každá jednotlivá část dat je uložena pouze jednou. [9] Nemůže tedy nastat situace, kdyby například nějaká entita byla uložena na dvou místech, třeba ve dvou různých proměnných a zároveň by se po změně v jedné z těchto proměnných tato změna neprojevila v té druhé. Z praktického pohledu jsou vlastně data uložena pouze jednou a všechny „manipulátory dat“ již pracují pouze s ukazateli na tato data.

3.3.5 HTML5 Canvas

Základním elementem v DOMu aplikace, který se bude starat o vykreslování grafu, je *canvas*, který byl zahrnut ve specifikaci HTML5. [10] Do prostoru, kde by měl být graf umístěn (viz návrh 3.1), bude vložena značka *<canvas>* tímto způsobem:

```
1 <canvas id="simulation" width="500" height="250">
2 </canvas>
```

Kód 3.2: HTML5 Canvas

Za povšimnutí stojí, že je tento element prázdný. To je proto, že o vykreslování grafu se bude starat JavaScript. Jediná věc, která se nastaví přímo do *canvasu* je výška a šířka našeho vykreslovacího plátna. V aplikaci bude rozměry v závislosti na aktuální velikosti okna prohlížeče zadávat komponenta, která bude mít na starosti vykreslení tohoto plátna. V javascriptovém kódu se potom canvasu dotážeme na jeho *2-D kontext*, do kterého je následně možné kreslit. Vykreslování probíhá voláním příslušných metod kontextu v definovaném pořadí. Před samotným vykreslením grafu se nejprve celé plátno překreslí tak, aby zobrazovalo pouze pozadí. Následují komponenty, které jsou „vzadu“ a následně ty, které mají být viděny celé. Aby byl graf zobrazen správně, bude se tedy vykreslovat v následujících vrstvách v tomto pořadí:

1. čisté pozadí,
2. hrany grafu,

3. uzly grafu,

4. prvky (předměty řazení),

5. hodnoty prvků (čísla).

3.3.6 D3

D3 je zkrácený název pro *Data-Driven Documents*, což je *javascriptová* knihovna umožňující snadnou manipulaci dokumentu v závislosti na datech. V tomto kontextu je jako dokument brán DOM webové stránky. Hlavní výhodou této knihovny je, že odstiňuje programátora od imperativního přístupu k datům. Namísto toho nabízí mnohem jednodušší deklarativní programování [11].

Problém s integrací

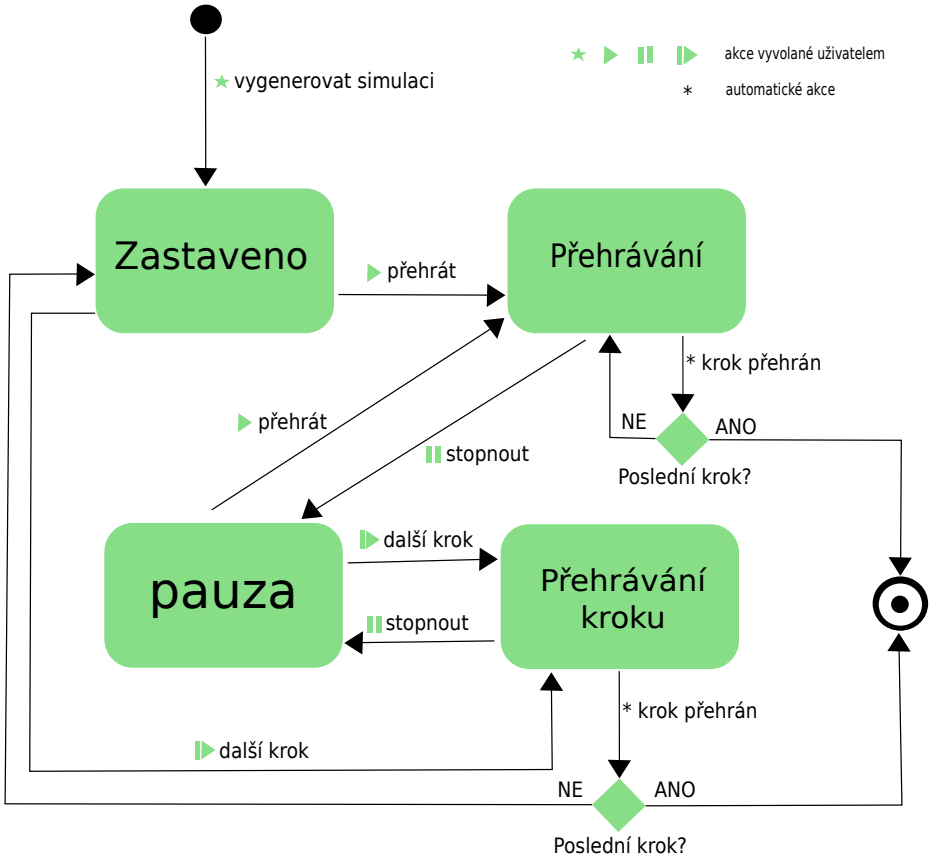
Knihovna D3 tedy manipuluje s DOMem, jenomže nad DOMem v aplikaci, která je napsána v Reactu, má plnou kontrolu právě React 3.3.3. Tím pádem nastává problém, protože D3 knihovna nemůže Reactu měnit DOM pod jeho rukama. Aby se změna projevila v aplikaci, je potřeba měnit pouze virtuální DOM, který je v Reactu. Naštěstí existuje řešení [12] jak zobrazovat grafy pomocí D3 knihovny v Reactu. Nicméně tato integrace jde dobře, pokud se jedná o samotné grafy, ale většinou pak ztroskotává ve chvíli, kdy potřebujeme do grafu zanést animace (nebo *přechody* v terminologii D3), protože v tuto chvíli již potřebujeme přímo manipulovat s dokumentem, což nám React nepovolí. Nicméně elegantním řešením, které bylo použito pro implementaci simulátoru v rámci této práce, je prostě nepoužívat D3 knihovnu na vykreslování grafu. O to se bude starat samotná komponenta napsaná v Reactu, která bude vykreslovat data pomocí JavaScriptu. Z knihovny D3 se potom využijí pouze funkce, které „časují“ průběh animace a funkce na generování přechodů mezi jednotlivými stavy. Jinými slovy nám D3 bude měnit pouze data, ale zodpovědnost za vykreslení dat do DOMu převezme React.

3.4 Stavový diagram simulace

Stavový diagram 3.9 zobrazuje přechody mezi různými stavy simulace. Každý přechod odpovídá akci, kterou buď vyvolal uživatel tím, že např. klikl na tlačítko „přehrát simulaci“, nebo akci, kterou vyvolá samotná simulace na konci přehrávání.

V diagramu jsou rozlišeny celkem čtyři stavy:

- Zastaveno – stav, ve kterém se aplikace nachází mezi jednotlivými kroky simulace. Zároveň se jedná o výchozí bod simulace.



Obrázek 3.9: Stavový diagram přehrávání simulace

- Přehrávání – v této fázi probíhá simulace. Aplikace zůstává v tomto stavu až do chvíle, než simulace skončí, nebo dokud není přerušena uživatelem.
- Pauza – jedná se o jakousi mezistanici mezi předchozím stavem a stavem následujícím.
- Přehrávání kroku – dalo by se zaměnit se stavem „přehrávání“, zde je ovšem rozlišeno, protože charakter přehrávání je jiný. Po každém kroku simulace se aplikace zastaví. Jedinou možností, jak se dostat do výchozího stavu „zastaveno“, je přes tento krok.

Není zde vyznačena akce, která by odpovídala „kroku zpět“. Tato akce by ovšem v grafu kopírovala trasu „kroku vpřed“, protože tyto akce jsou

prakticky totožné, akorát jedna akce bere předchozí stav simulace a druhá následující.

Kapitola 4

Implementace

Implementace aplikace byla rozdělena do dvou zcela nezávislých celků, které byly programovány v tomto pořadí:

1. Implementace prezentační vrstvy – naprogramování uživatelského rozhraní a simulace.

2. Implementace simulátoru a jednotlivých algoritmů.

Pro implementaci simulátoru a algoritmů je potřeba dopředu znát přesnou podobu dat, se kterými se bude pracovat. Určit nejvhodnější formát, kterými budou kroky simulace a jednotlivé stavy reprezentovány ovšem není snadný úkol. Z tohoto důvodu byl zvolen následující postup.

Nejprve se naimplementuje prezentační vrstva včetně veškeré grafiky takovým způsobem, aby simulátor běžel přesně podle specifikace. Přičemž budou využita testovací data, která budou simulátoru předložena a která budou dopředu nadefinována. Během implementace se pak testovací data doplní podle potřeby. Ve výsledku pak dostaneme strukturu dat (podobu stavů simulace a struktury grafů), kterou bude generátor simulací vracet na výstup. Potom, co budeme vědět, jaký přesně má simulátor mít výstup, můžeme pokračovat v jeho implementaci.

4.1 Implementace prezentační vrstvy

Prezentační vrstvou jsou myšleny nejen komponenty, které se starají o interakci s uživatelem, ale také formát, který bude mít zobrazovaný graf a simulace. Ve schématu aplikace 3.8 představenému v návrhu 3 se jedná o první dvě vrstvy shora.

4.1.1 Datová reprezentace

V této sekci je popsáno, jaký formát mají data, která má generovat simulátor a která budou aplikací prezentována do podoby simulace. Základní entity tohoto datového modelu jsou:

- prvky k seřazení,
- graf simulace,
- data reprezentující stavy simulace.

Prvky k řazení

Prvky, které bude simulace řadit, jsou vlastně čísla, jejichž hodnota pak určuje výslednou pozici, ve které se bude dané číslo nacházet v seřazené posloupnosti. Datovou reprezentací by tak mohlo být pole čísel. Nicméně počet prvků není konstantní, protože během simulace dochází k jejich duplikaci a opětovnému mazání. Přičemž je důležité si pamatovat přesně, které číslo bylo zkopírováno nebo smazáno a zároveň je potřeba rozlišovat čísla, která mají stejnou hodnotu.

Z tohoto důvodu jsou prvky reprezentovány jako pole objektů, kde každý prvek má unikátní identifikační číslo a hodnotu. Později při vykreslování jsou každému prvku přiřazeny souřadnice a další informace týkající se například animací a efektů.

```
1  export const TestValues = [
2    {
3      id: 0,
4      value: 46,
5    },
6    {
7      id: 1,
8      value: 77
9    },
10   {
11     id: 2,
12     value: 51
13   },
14 ...
15 ];
```

Kód 4.1: Reprezentace prvků

Reprezentace grafu

Graf, který je vykreslován v simulaci, má následující hodnoty:

- **Hrany** jsou reprezentovány jako pole dvojic. Dvojice se skládá ze dvou identifikátorů, které označují jaké dva uzly grafu jsou propojeny hranou. Protože jsou hrany jakožto komunikační kanál duplexní, nezáleží na pořadí, v jakém jsou uzly ve dvojici zapsány.
- **Uzly** jsou reprezentovány jako pole objektů, kde každý objekt má následující atributy:
  - *Počet hodnot* je číslo, které udává, kolik prvků je v uzlu „standardně“, tedy po redukci provedené po lokálním seřazení prvků. Skutečný počet prvků může být v uzlu pouze větší, nebo roven této hodnotě.
  - *Hodnoty* jsou polem čísel. Každé číslo reprezentuje identifikátor hodnoty, který jednoznačně určuje prvek nacházející se v poli hodnot 4.1.1. Toto pole tedy reálně udává, jaké prvky jsou v uzlu přítomny.
  - *Souřadnice* udávají umístění v grafu a mají relativní charakter. To znamená, že jejich hodnoty pouze odlišují, v jaké pozici se vzájemně nacházejí jednotlivé uzly grafu.
- **Velikost prvku** je číslo udávající relativní šířku jednoho prvku k řazení a tím pádem ovlivňující i výslednou velikost každého uzlu v grafu.

```
1  export const TestGraphQ2 = {
2    valueSizeInPixels: 50,
3    nodes: [{
4      x: 50,
5      y: 50,
6      valuesCount: 3,
7      values: [7, 3, 9]
8    },{
9      x: 250,
10     y: 50,
11     valuesCount: 3,
12     values: [8, 2, 1]
13   },{
14     x: 50,
15     y: 250,
16     valuesCount: 3,
17     values: [0, 10, 6]
18   },{
19     x: 250,
20     y: 250,
21     valuesCount: 3,
22     values: [4, 11, 5]
```

```
23    }],
24    edges: [[0,1],[0,2],[1,3],[2,3]],
25  }
```

Kód 4.2: Datová reprezentace dvourozměrné krychle pro testování

Kroky simulace

Jelikož je kroků simulace poměrně hodně, je vhodné využít minimalistickou datovou reprezentaci. Jediné údaje, které potřebujeme uchovávat jsou:

1. Na kterých uzlech jsou přítomny které prvky.
2. V jakém jsou tyto prvky pořadí.
3. Jaký je jejich celkový počet.

Formát jediného kroku simulace byl zvolen tak, že se jedná o pole vnitřních stavů uzlů přesně v takovém pořadí, v jakém jsou brány uzly konkrétního grafu. Vnitřní stav uzlu je potom reprezentován polem identifikátorů prvků.

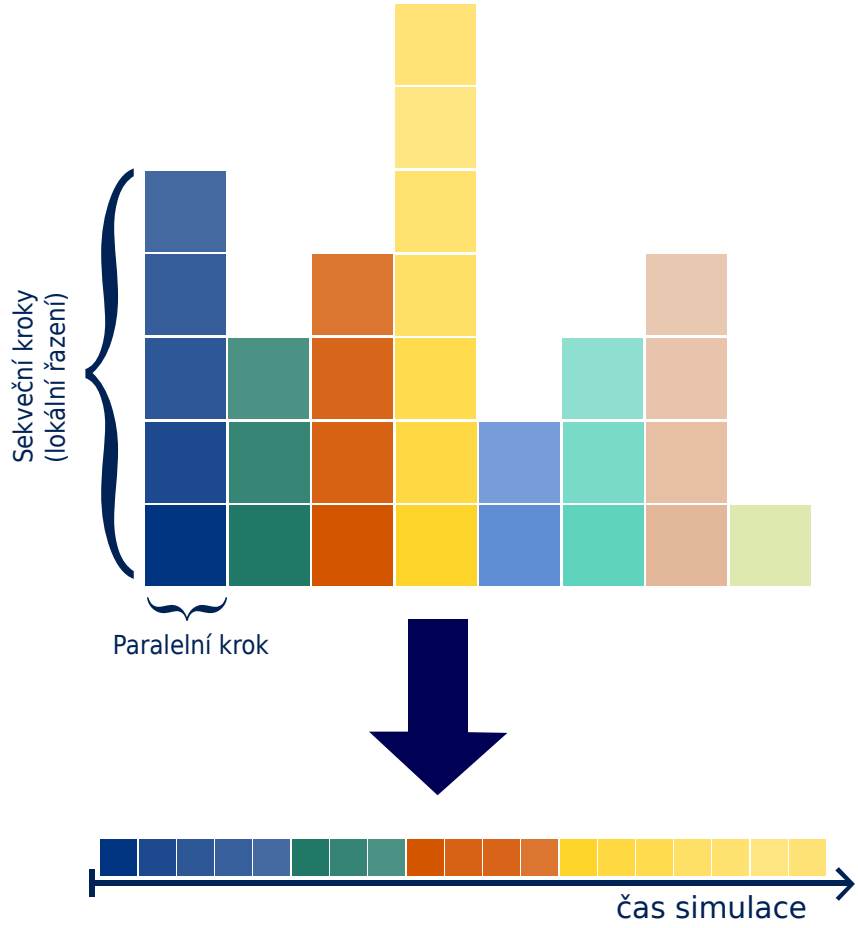
```
1  const TestSequentialSteps = [
2    [
3      [7, 3, 9],
4      [8, 2, 1],
5      [0, 10, 6],
6      [4, 11, 5],
7    ],
8    [
9      [3, 7, 9],
10     [8, 1, 2],
11     [0, 6, 10],
12     [4, 5, 11],
13   ],
14 ];
```

Kód 4.3: Struktura dvou po sobě jdoucích kroků

Simulace

Základem simulace je opět pole, které obsahuje seznam po sobě jdoucích paralelních kroků simulace. Každý tento paralelní krok je reprezentován polem sekvenčních kroků představujícím průběh lokálních řazení na všech uzlech grafu. Reprezentace tohoto pole je vidět v ukázce kódu 4.3.

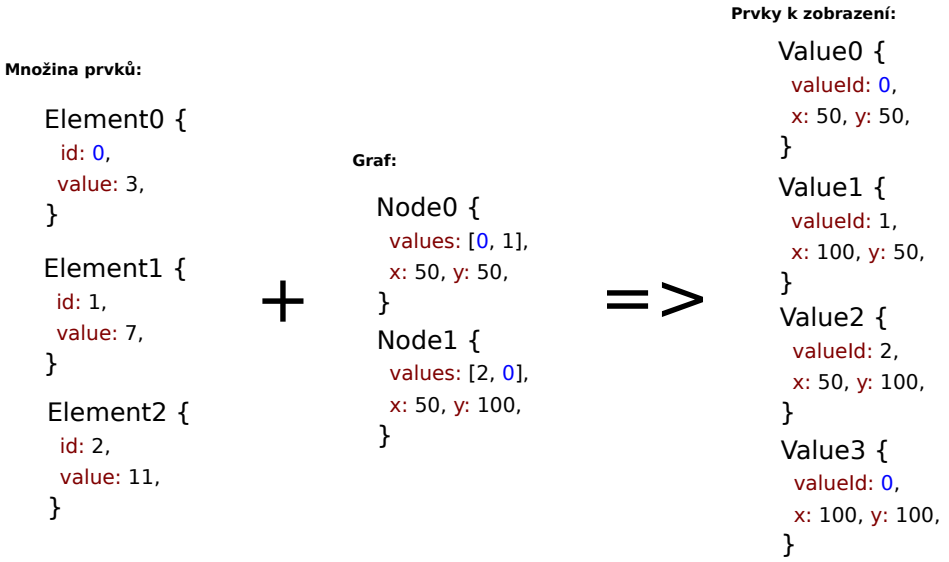
Simulace probíhá tak, že se postupně zobrazují stavy sekvenční. Když simulace narazí na poslední sekvenční krok, přepne se na další paralelní krok,



Obrázek 4.1: Datová reprezentace simulace

kde začne od začátku. Celý proces pokračuje, dokud se neprojdou všechny dostupné stavy. Tento proces je zobrazen na obrázku 4.1.

Tento dvourozměrný model byl zvolen proto, že při implementaci jednotlivých algoritmů uvažujeme pouze paralelní kroky. Lokální řazení se pak děje zcela nezávisle na každém uzlu a do implementace paralelního řazení vůbec nezasahuje. Po každém paralelním kroku se zavolá lokální řazení na všech uzlech grafu, přičemž toto sekvenční řazení může být implementováno jako libovolný řadící algoritmus.



Obrázek 4.2: Transformace prvků v grafu pro zobrazení

4.1.2 Příprava prvků pro zobrazení

Prvky datového modelu (viz Datová reprezentace 4.1.1) nelze zobrazovat přímo. Je potřeba navzájem tyto prvky zkombinovat. Jako příklad vezměme prvky k seřazení. Těch se nachází v simulaci určité omezené množství, ale během simulace se jejich počet neustále mění. Při každém paralelním kroku dochází k jejich kopírování a je důležité si udržovat informaci o tom, které kopie náleží ke kterému prvku.

Jako řešení byl zvolen přístup, že se do původní množiny vůbec nezasahuje. Ta je stále konstantní. Namísto toho se využije mapovací funkce, která na vstup vezme tuto množinu původních prvků a aktuální stav grafu. V uzlech grafu jsou potom rozmístěny identifikátory hodnot, které se v nich nachází. Tato mapovací funkce potom na základě těchto vstupních hodnot vypočítá novou množinu hodnot, jejíž velikost již odpovídá skutečnému množství hodnot v simulaci. Zároveň již každý prvek obsahuje souřadnice v závislosti na tom, v kterém uzlu grafu se zrovna nachází.

V tento okamžik již máme k dispozici všechny potřebné údaje k tomu, abychom mohli prvky zobrazit. Výhodou je rovněž skutečnost, že původní množina zůstává nedotčena a máme tak k dispozici před každým vykreslením stejné hodnoty. Simulace se tedy může spoléhat na to, že pracuje se stále stejnou množinou dat. Celý tento postup je zobrazen na obrázku 4.2.

4.1.3 Zobrazení

Dalším důležitým objektem v simulátoru je **kamera**. Doposud měly všechny souřadnice pouze relativní charakter a z jejich hodnot se dalo usuzovat pouze to, jak jsou jednotlivé entity uspořádány vzhledem k sobě navzájem. Kamera potom určuje skutečnou polohu těchto entit v zobrazovaném prostoru.

Protože má každý graf jiné rozměry, je vhodné velikost grafu přizpůsobit rozměrům okna prohlížeče. Po vygenerování grafu je potřeba, aby se graf zobrazil celý. Nicméně dopředu počítáme s tím, že bude možné si určitou část grafu přiblížit. Z tohoto důvodu musí kamera obsahovat hodnoty, které jsou zobrazeny v ukázce 4.4. Jedná se o tyto položky:

- *Zoom* – což je desetinné číslo, které udává přiblížení kamery. Po vygenerování simulace je toto číslo automaticky spočítáno v závislosti na rozměrech daného grafu. Zároveň lze kdykoliv v průběhu simulace hodnotu tohoto čísla měnit. Typicky uživatel použije rolovací kolečko myši k přiblížení nebo oddálení grafu.
- *Souřadnice* – zde se již jedná o skutečné souřadnice v pixelech, které má kamera. Jedná se o výchozí bod, v závislosti na kterém se spočítají souřadnice zobrazovaných objektů. Pozici tohoto bodu může rovněž uživatel měnit. Toto mu společně s přiblížením umožňuje zobrazovat konkrétní menší části grafu.

```
1 const camera = {
2   zoom: 1.0,
3   x: 350,
4   y: 0,
5 }
```

Kód 4.4: Kamera

Vykreslování

Jak již bylo zmíněno v sekci věnující se zvoleným technologiím 3.3.5, vykreslování všech entit probíhá pomocí JavaScriptu přímo do *canvasu* voláním metod jeho *kontextu*. Toto vykreslování probíhá podobným způsobem, jako bychom kreslili na plátno ručně. Nejdříve si zvolíme barvu a „sílu štětce“, to se v kódu nastavuje pomocí *context.fillStyle* resp. *context.strokeStyle* a dále již definovanou posloupností příkazů vedeme jednotlivé tahy. Každý tah začíná zavoláním *context.beginPath()* a uzavírá se pomocí *context.closePath()*.

Pro názornost je v ukázce kódu 4.5 zobrazeno vykreslování obdélníku se zaoblenými rohy, který se používá pro vykreslení uzlu grafu. Pro každou entitu určenou k vykreslení je podobným způsobem definována funkce, která má na vstupu kromě kontextu ještě rozměry a souřadnice zobrazovaného objektu.

```
const drawRoundedRectangle =
  (context, x, y, w, h, r) => {
    if (w < 2 * r) r = w / 2;
    if (h < 2 * r) r = h / 2;
    context.beginPath();
    context.moveTo(x+r, y);
    context.arcTo(x+w, y,   x+w, y+h, r);
    context.arcTo(x+w, y+h, x,   y+h, r);
    context.arcTo(x,   y+h, x,   y,   r);
    context.arcTo(x,   y,   x+w, y,   r);
    context.closePath();
    return context;
}
```

Kód 4.5: Vykreslení obdélníku

4.1.4 Průběh simulace

V této fázi máme naimplementováno zobrazování grafu, který aplikaci předáme a jehož strukturu máme předem nadefinovánu. Ještě předtím, než bychom se mohli pustit do implementace generátoru simulací a jednotlivých algoritmů, je potřeba naimplementovat průběh simulace na našem grafu tak, aby simuloval všechny situace, které mohou během přehrávání nastat. Toto nám pomůže dodefinovat, jakou přesně mají mít simulační data strukturu a vlastnosti. Je tedy potřeba naimplementovat průběh alespoň těchto specifických situací:

- Výměna dvou prvků na uzlu grafu – případ, kdy dochází k lokálnímu řazení na procesoru. Probíhá na více uzlech grafu zároveň.
- Zkopírování všech prvků ve dvou sousedních uzlech a jejich vzájemná výměna.
- Odstranění několika prvků z uzlu – toto je dobré zobrazit jako plynulou animaci.

D3-timer

Pro průběh časování celé simulace se využije funkce *timer* z knihovny D3 (viz sekce 3.3.6). Tato funkce se sama stará o to, aby byl průběh animace plynulý. Na vstup bere funkci, která je neustále volána v časových intervalech s parametrem určujícím, kolik času uběhlo od začátku simulace. Toto volání probíhá až do chvíle, než se zavolá funkce *stop()* na objekt, který tento časovač obsahuje. V ukázce 4.6 je vidět kostra základní komponenty, která se o průběh animace stará.

```
1  import React from 'react';
2  import { timer } from 'd3-timer';
3
4  class Timer extends React.Component {
5    // ...
6    play () {
7      this.transition = timer(elapsed => {
8          // Implementation of animations
9          // in dependence on elapsed time.
10       });
11   }
12 }
```

Kód 4.6: Hlavní komponenta řídící animace

D3-interpolate

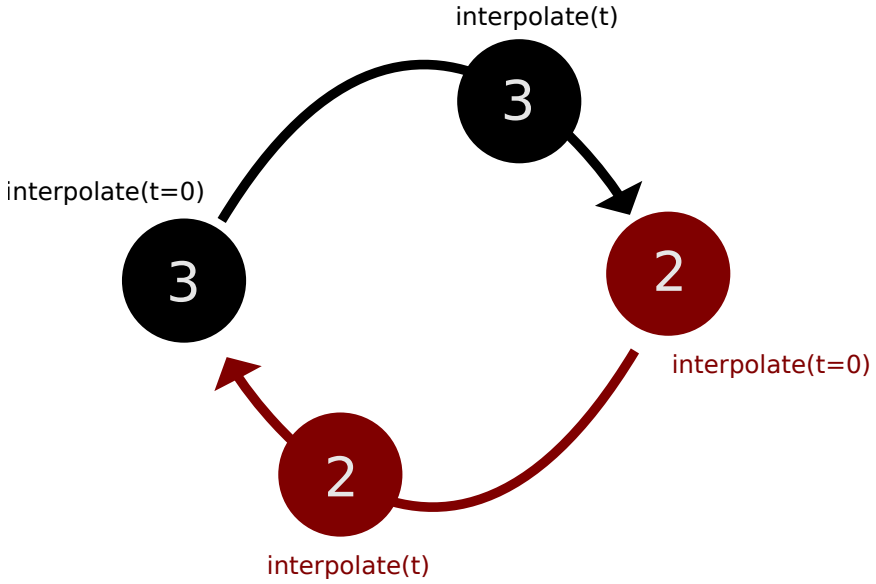
Další mocnou množinou funkcí z knihovny D3 jsou funkce, které se starají o *interpolaci* objektů v čase. Simulace má vždy k dispozici pouze aktuální krok simulace a krok následující. Pohyb mezi těmito dvěma stavy je potřeba dopočítávat podle aktuálního času, který uplynul od spuštění časovače (typicky na začátku každého kroku).

Ke všem objektům, které se v simulaci aktuálně pohybují, přiřadíme funkci, která vrátí jejich pozici v čase. K tomu se dají využít funkce *interpolateNumber* a *interpolateObject*, které mají dva parametry – počáteční a koncový stav. Návratovou hodnotou je opět funkce, která řídí změnu objektu z času $t = 0$ do času $t = 1$, který je brán jako koncový.

Protože se nejčastěji prvky pohybují v simulaci proti sobě navzájem, je vhodné pro lepší přehlednost jejich pohyb nevést po přímce. Namísto toho se výměna prvků provede tak, aby se prvky navzájem „vyhnuly“. K tomu má knihovna D3 k dispozici spoustu funkcí, které dokáží přechod vést jiným než lineárním způsobem. Konkrétně byla pro situaci, kdy si prvky prohazují své pozice, využita funkce *d3.easeSin(t)*, díky které je možné pohyb prvků vést po sinusoidě. Celá tato situace je zobrazena na obrázku 4.3, kde jsou vidět dva prvky v počátečním čase a v konkrétním čase t.

4.2 Implementace simulátoru

Nyní se aplikace nachází ve stavu, kdy je schopna zobrazit kompletně graf, přehrát simulaci s předloženými daty a zároveň splňuje všechny ostatní požadavky, které byly kladeny na uživatelské rozhraní a ovládání. Je definován kompletní soubor dat, který je potřeba pro běh simulace. V této fázi je tedy možné se zabývat implementací jednotlivých algoritmů. Napřed ale bude nutné



Obrázek 4.3: Interpolace dvou prvků v čase, které si vyměňují pozici

navrhnout jakési rozhraní, pomocí kterého bude nynější prezentační vrstva algoritmům rozumět.

4.2.1 Generátor algoritmů

Generátoru jsou předány od uživatele vstupní parametry, které ovlivňují generované stavy výsledné simulace. Následuje vygenerování dalších potřebných dat, která se dají vypočítat ze zadaných parametrů a která zcela nezávisí na konkrétním algoritmu. Konkrétně se jedná například o prvky simulace. Uživatel zadal parametry jako je počet prvků na uzel a velikost generovaného grafu. Potom následují tyto kroky, které se provedou vždy stejně bez ohledu na to, jaký algoritmus je simulován:

1. Z předložených rozměrů se vypočítá, kolik celkem prvků bude simulace obsahovat.
2. Vygeneruje se pole prvků o potřebné velikosti.
3. Vytvoří se nové pole identifikátorů těchto prvků, ve kterém budou rozmístěny v náhodném pořadí. Při generování grafu pak každý algoritmus bude přidělovat prvky jednotlivým uzlům z tohoto pole.

Potom již následují kroky, které jsou specifické pro každý algoritmus:

1. Vygenerování grafu z předložených parametrů a vygenerovaných hodnot.

2. Definování funkce, která simuluje řazení na vygenerovaném grafu.

Funkce, která simuluje řazení, dodržuje schéma podle ukázky 4.7. Na vstupu bere vždy vygenerovaný graf, vygenerované hodnoty a ostatní vstupní parametry. Na výstup vrací pole reprezentující stavy v jednotlivých krocích simulace (viz Kroky simulace 4.1.1). Druhý výstup reprezentuje podobné pole, které obsahuje informace o tom, v kterém kroku jsou které hrany aktivní (tedy pomocí kterých uzly komunikují). Toto pole bylo přidáno později, když se přišlo na to, že je jednodušší, když se zodpovědnost za rozhodnutí, která hrana je aktivní a která nikoliv, přenese na každý algoritmus, který tuto informaci může ukládat v průběhu počítání dalších stavů, namísto dopočítávání na základě rozdílů mezi předchozím a následujícím stavem během zobrazování.

```
1 const simulate = (graph, options, values) => {
2   // Every algorithm implements this code
3   // ...
4   return {
5     values: implementedStates,
6     edges: activeEdges,
7   };
8 }
```

Kód 4.7: Funkce pro generování simulace

4.3 Implementace algoritmů

Implementaci algoritmů je dobré realizovat co nejvíce obecně, tedy pomocí funkcí, které se budou dát použít ve všech algoritmech. Jinými slovy si nadefinujeme sadu funkcí, která bude implementovat elementární instrukce prováděné během běhu u každého algoritmu. Díky tomuto přístupu se následně jednotlivé algoritmy budou v implementaci lišit v co nejmenším počtu řádků.

4.3.1 Základní Ramda funkce

V dalších částech této sekce budou vždy uvedeny konkrétní ukázky kódu. Snaha byla ovšem kód psát co nejvíce funkcionálně (viz technologie 3.3).
Z toho důvodu zde představím krátce základní funkce, které se vyskytují ve funkcionálním programování a které jsou v ukázkách použity z knihovny Ramda:

- **Map** – v kódu se nevyskytují žádné cykly, které známe z imperativního programování *(for, while)*. Namísto toho, pokud potřebujeme iterovat nade všemi prvky nějakého pole (nebo i objektu), použijeme mapovací funkci. Tato funkce nám nad každým prvkem zavolá námi definovanou funkci a výsledek vrátí jako nové pole (nebo objekt). Tím máme mimo

jiné zaručeno, že nedojde ke změně vstupních hodnot, takže celý tento proces je neměnný (*immutabilní* viz funkcionální přistup 3.3.2).

- **Filter** – funkce, která podobně jako Map spustí námi definovanou funkci nad každým prvkem. Na výstup ovšem vrátí pouze ty prvky, na které tato funkce vrátí pozitivní odpověď *(true)*. Jak již tedy název napovídá, je tato funkce vhodná jako filtr, který z množiny odstraní nepotřebné prvky.
- **Reduce** – využívá se, pokud potřebujeme provést operaci nade všemi prvky, při které potřebujeme u každého dalšího prvku počítat s hodnotou spočítanou v předchozí iteraci. Jako typický příklad můžeme uvažovat funkci, která spočítá součet všech hodnot v poli.
- **Range** – vrací pole hodnot v daném rozsahu. Většinou se používá, pokud potřebujeme například pracovat s indexy. V takovém případě zavoláme funkci Map nad polem vytvořeným pomocí funkce Range.
- Dalšími funkcemi, které mohou být v ukázkách uvedeny, jsou pomocné funkce pro práci s poli:
  - **Head**, která vrací první prvek pole.
  - **Last** vracející poslední prvek pole.
  - **Init**, která vrací všechny prvky kromě posledního.
  - **Tail** vracející všechny prvky kromě prvního.

4.3.2 Společné funkce

V následujících odstavcích představím stručně některé funkce společné všem algoritmům, které byly použity pro implementaci. Nicméně ukázky zde uvedené mají pouze demonstrativní charakter a od skutečné implementace se mohou trochu lišit, zejména pak v názvech konstant a funkcí.

Kopírování prvků

Každý algoritmus využívá operaci *copyAndExchange*, která vezme dva sousední uzly grafu a zkopíruje jejich prvky. Zkopírované prvky uzlu potom uloží do paměti sousedního uzlu.

```
1 const copyAndExchange = ([node1, node2]) =>
2   [[...node1, ...node2], [...node2, ...node1]];
3
4 copyAndExchange([[1,2,3], [4,5,6]])
5 // => [[1, 2, 3, 4, 5, 6], [4, 5, 6, 1, 2, 3]]
```

Kód 4.8: Funkce pro výměnu prvků mezi uzly

Jak je vidět v ukázce 4.8, zkopírované prvky od souseda se vždy připojí k těm stávajícím na konec.

Lokální řazení

Po každé výměně prvků zpravidla přichází na řadu seřazení prvků na každém uzlu. Jelikož je tento postup součástí simulace, je potřeba celý postup řazení ukládat. Funkce *generateLocalSorts* tedy generuje pole představující stavy jednotlivých prvků až do konečného stavu, kdy jsou na každém uzlu prvky seřazeny.

```
1  const generateLocalSorts = (nodes, values,
       seqSortAlgorithm) => {
2    // 1. Perform local sort on every node
3    const sorts = R.map(n => seqSortAlgorithm(n, values
       ), nodes);
4    // 2. Adjust the length
5    return R.map(n => R.map(a => {
6        if (n >= a.length-1)
7          return R.last(a);
8        else
9          return a[n];
10     }, sorts), R.range(0, getMaxSize(sorts)));
11 }
```

Kód 4.9: Lokální řazení

Jak je vidět z ukázky 4.9, tak lokální řazení probíhá ve dvou krocích:

- Nejprve se zavolá sekvenční řadící algoritmus na všechny uzly. Tento algoritmus je funkci *generateLocalSorts* předán parametrem, nicméně nejedná se o klasický řadící algoritmus, ale o upravený, který vrací v poli každý krok, který byl prováděn. Výsledku řazení tak odpovídá poslední prvek z navráceného pole. Každý sekvenční řadící algoritmus, který by byl do aplikace přidáván, tak musí být upraven, aby generoval všechny stavy řazení.

- Protože řazení na každém uzlu může skončit v různém počtu kroků, je potřeba upravit velikost všech těchto polí reprezentujících lokální řazení, aby se jejich délka shodovala s nejdelším polem. To vlastně znamená, že se u kratších polí bude opakovat jejich poslední stav až do délky nejdelšího pole. Tímto do simulace zaneseme „čekání“ uzlů na to, až všechny ostatní uzly setřídí své vlastní prvky.

Příklad toho, jak probíhá volání funkce pro lokální řazení, je vidět v ukázce 4.10. Pro lepší názornost jsou v ukázce identifikátory prvků shodné s jejich

hodnotami. Lze tak snadno z výstupu funkce vyčíst, jakým způsobem probíhá samotné řazení.

```
generateLocalSorts(
  [[3,4,2], [0,3,1]], // states of nodes
  [{id: 0, value: 0}, {id: 1, value: 1},
   {id: 2, value: 2}, {id: 3, value: 3},
   {id: 4, value: 4}], // array of values
  selectionSort // sequential sorting algorithm
)
// => [[[3, 4, 2], [0, 3, 1]],
//     [[2, 4, 3], [0, 1, 3]],
//     [[2, 3, 4], [0, 1, 3]]]
```

Kód 4.10: Použití lokálního řazení

Redukce

Potom, co jsou prvky zkopírovány a seřazeny na každém uzlu, se nacházíme ve stavu, kdy se v grafu vyskytuje celkově dvojnásobný počet prvků. Je tedy potřeba tento počet zredukovat a odstranit duplicitní prvky.

K tomuto účelu může posloužit funkce *removeHalf*, které předáme prvky uzlu a příznak, jestli chceme menší nebo větší polovinu. Rozhodování, kterému uzlu necháme jeho větší a kterému menší část prvků, už je zcela v režii každého algoritmu. Zároveň nemusíme řešit situaci, kdy by počet prvků byl lichý, což by do implementace zanášelo problém, jestli zahodit menší, nebo větší část, protože před použitím této operace bude počet prvků vždy sudý. To z toho důvodu, že této operaci předchází rozšíření počtu prvků o prvky ze sousedního uzlu, který má vždy stejný počet prvků jako cílový uzel, takže počet prvků je před použitím redukce vždy dvojnásobný, tedy vždy sudý.

```
const removeHalf = (node, smaller = true) =>
  smaller ?
    R.head(R.splitAt(R.length(node)/2, node))
  :
    R.last(R.splitAt(R.length(node)/2, node))

removeHalf([1, 2, 3, 4]) // => [1, 2]
removeHalf([1, 2, 3, 4], false) // => [3, 4]
```

Kód 4.11: Redukce prvků uzlu

4.3.3 Sudo-liché řazení na 1-D mřížce

V posledních částech této sekce se zabývám implementací jednotlivých algoritmů. Nicméně implementace je docela složitější a tak jsou u každého algoritmu uvedeny pouze „úryvky“. Tyto části kódu byly zjednodušeny na takovou

míru, aby pouze demonstrovaly, co se přibližně děje v průběhu generování simulace. Zároveň na začátku každého algoritmu dochází nejprve k lokálnímu seřazení prvků a tak se u ukázek uvažuje, že tento stav je výchozím bodem.

Generování grafu

Každý algoritmus pracuje se specifickou formou grafu, takže musí každá implementace algoritmu obsahovat i funkci, která tento graf vygeneruje. Generování 1-D mřížky je v tomto případě snadné, uzly grafu jsou seřazeny v poli podle indexu a hrany jsou polem dvojic sousedních indexů.

Simulace

Pro Sudo-liché řazení je typické, že v každém kroku spolu komunikují uzly pouze přes sudé nebo liché hrany. Z tohoto důvodu byla naimplementována pomocná funkce *evenOrOdd*, které na vstup předáme číslo označující aktuální krok a která vrací pole indexů uzlů, které si vymění hodnoty se sousedem vpravo.

```
1  // For every step do:
2  const states = map(stepNumber => {
3    // 1. For every even/odd node do:
4    map(i => {
5      // Copy values and exchange with neigbour
6      copyAndExchange(nodes[i], nodes[i+1])
7      ...
8    }, evenOrOdd(nodes, stepNumber))
9    ...
10   // 2. Generate local sorts
11   const sortedNodes = sortNodesLocal(nodes, values)
12   ...
13   // 3. For every even/odd node do:
14   const reducedNodes = map(i => {
15     removeLargerHalf(nodes[i])
16     removeSmallerHalf(nodes[i+1])
17   }, evenOrOdd(nodes, stepNumber))
18   ...
19   // 4. Consctruct and push state
20   return constructedState;
21
22 }, range(0, getNumberOfSteps(n)))
```

Kód 4.12: Sudo-liché řazení

V ukázce kódu 4.12 jsou v komentářích čísly označeny tyto kroky algoritmu:

1. V každém kroku se neprve provede výměna prvků mezi uzly, které jsou označeny pomocí funkce *evenOrOdd*.
2. Vygenerují se stavy, reprezentující lokální řazení na všech uzlech.
3. Fáze redukce probíhá tak, že každý uzel vlevo (ten s menším identifikátorem), si ponechá menší polovinu prvků a jeho soused napravo si ponechá tu větší.
4. Dalším krokem je sestavení pole představujícího aktuální paralelní krok simulace. Vygeneruje se stav pro uzly a stav pro hrany.

4.3.4 ShearSort na 2-D mřížce

Pro implementaci ShearSortu se dá využít již naimplementovaného sudo-lichého řazení, protože v každém paralelním kroku simulace probíhá toto řazení nad určitými podgrafy 2-D mřížky (buď sloupce, nebo řádky).

Generování grafu

Vygenerování uzlů není zase nic složitého, jedná se prakticky o 2-D pole, které není problém převést na klasické 1-D pole potřebné pro řízení simulace, nicméně je potřeba myslet na to, že uzly jsou seřazeny tzv. „hadovitým“ způsobem, jako je zobrazeno na obrázku 2.4 v sekci Analýza algoritmů 2.3.

Postup generování grafu pro ShearSort je tedy takový, že se vygeneruje obyčejné 2-D pole podle zadaných rozměrů. Potom se nad tímto polem zavolá mapovací funkce, která jej projede řádek po řádku a každý druhý vrátí v obráceném pořadí.

Simulace

Implementace Shearsortu už začíná být komplikovanější. Shearsort se sice skládá z již hotového sudo-lichého řazení, nicméně právě proto, že jsou řádky grafu uspořádány střídavými směry, je nutné naimplementovat řadu funkcí, které z našeho grafu sestaví potřebný podgraf, na který můžeme použít Sudoliché řazení. Je potřeba mít na paměti, že v datové reprezentaci jsou uzly uloženy do obyčejného 1-D pole. Mezi tyto pomocné funkce patří následující:

- *horizontalSubgraph* – extrahuje řádek na základě zadaných parametrů, jako je index řádku, počet hodnot v řádku a seznamu uzlů.
- *verticalSubgraph* – totéž jako předchozí, akorát namísto řádku vrací sloupec.
- *invertArray* – jednoduchá funkce pro obrácení pořadí uzlů v předaném poli.

```
// Shearsort
const r = reduce((previous, actual, index) => {
  if (isOdd(index)) {
    return doSnaking(...)
  } else {
    return doVertical(...)
  }
}, doSnaking(/*initial*/), range(0, numberOfSteps))
```

Kód 4.13: Shearsort

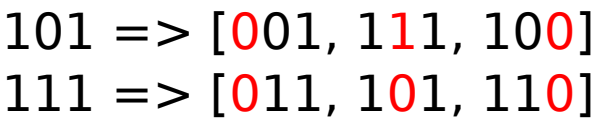
V průběhu algoritmu se stejně jako u toho předchozího střídají dva postupy na základě toho, jestli se jedná o sudý, nebo lichý krok výpočtu. Tyto postupy jsou rozděleny do funkcí *doVertical*, která seřadí v grafu všechny sloupce směrem dolů, a *doSnaking*, která seřadí všechny řádky střídavým směrem a která je popsána v ukázce 4.14.

```
const doSnaking = (nodes, values, m, n) => {
  // 1. For every row do:
  map(mIndex => {
    // 2. Get subgraph
    const subgraph = horizontalSubgraph(mi, n, nodes)
       ;
    // 3. Perform Even-odd sort
    const simulated = simulateEvenOdd(
       isOdd(mIndex) ? invertArray(subgraph) :
          subgraph,
       values, m);
    // 4. Is it odd row? Invert result.
    return isOdd(mIndex) ? invertArray(simulated) :
       simulated;
  }, range(0, n))
  return {values, edges}
}
```

Kód 4.14: Hadovité řazení

Popis funkce *doSnaking*:

1. V této funkci se iteruje přes všechny řádky.
2. Vytáhne se z grafu aktuální řádek.
3. Nad aktuálním řádkem se provede Sudo-liché řazení. Pokud je aktuální krok lichý (čísluje se od nuly a ta se bere jako sudá), tak se nejprve řádek obrátí pomocí funkce *invertArray*.
4. Pokud byl podgraf obrácen v předchozím kroku, obrátí se při návratu zpět.



Obrázek 4.4: Generování sousedů v hyperkrychli

4.3.5 Bitonický MergeSort na hyperkrychli

Na hyperkrychli již nastává zcela odlišná situace. Uzly je dobré číslovat binárně, neboť jejich binární zápis přesně udává jejich umístění v grafu. Pokud projíždíme binární zápis identifikátoru od prvního bitu zprava, tak každá jednička nám udává, že máme při vykreslování daný uzel posunout v této dimenzi. Sousední uzly se potom liší pouze v jednom bitu, jehož pořadí nám udává, v jaké dimenzi se sousední uzly liší. Každý uzel má tedy tolik sousedů, kolika dimenzionální je graf.

Generování grafu

V ukázce kódu 4.15 je vidět pomocná funkce, která generuje všechny sousedy pro předložený identifikátor. Na obrázku 4.4 je pak znázorněno, jak se jednotlivé uzly od sebe liší.

```
1  const generateNeighbours = (dimensions, number) => {
2    return R.map((n) => {
3      if (number & Math.pow(2, n))
4        return (~(Math.pow(2, n)) & number)
5      else
6        return ((Math.pow(2, n)) | number);
7    }, R.range(0, dimensions));
8  }
9
10 generateNeighbours(3, 5) // => [4, 7, 1]
```

Kód 4.15: Generování sousedů

Tato funkce se používá hlavně pro generování hran. Uzly jsou opět uloženy v jednorozměrném poli.

Simulace

Řazení na hyperkrychli probíhá v pořadí, které generuje funkce v ukázce 4.16. Průběh probíhá v několika kolech tak, že se v každém kole postupně snižuje dimenze, po poslední nulté dimenzi se přejde do dalšího kola. Každé následující

kolo začíná o jednu dimenzi výš než předchozí. Začíná se v nulté dimenzi a končí se kolem, které obsahuje všechny dimenze.

```
const generateSteps = (dimensions) =>
  R.reduce((previous, step) =>
    [...previous, ...step], [],
    R.map(round =>
      R.map(dim => dim,
        R.reverse(R.range(0, round+1))),
          R.range(0, dimensions)))

generateSteps(4)
// => [0,   1, 0,   2, 1, 0,   3, 2, 1, 0]
```

Kód 4.16: Generování pořadí dimenzí

Další náročnější operací oproti předchozím algoritmům je určení, které uzly si po výměně a lokálním seřazení prvků nechají menší polovinu dat a které tu větší. Kvůli tomu byla naimplementována rekurzivní funkce, která na vstupu bere hloubku rekurzivního stromu (odpovídá počtu dimenzí) a označení dimenze, ve které se algoritmus aktuálně nachází. Na výstup potom vrací binární pole, kde jednička značí, že si uzel (na stejné pozici v poli, na jaké se nachází jednička) nechává větší polovinu a naopak.

```
const generateAscends = (depth, final, asc = 1) => {
  if (depth == final) {
      return R.map(() => asc, R.range(0, Math.pow(2,
         final)))
    }
    else
      return [...generateAscends(depth-1, final, 0)
         ,...generateAscends(depth-1, final, 1)]
}

generateAscends(3,1) // => [0, 0, 1, 1, 0, 0, 1, 1]
```

Kód 4.17: Generování pole určujícího, které uzly mají větší a které menší polovinu prvků.

Kostrou celého algoritmu je potom tento proces:

- Pro každou dimenzi d z pole získaného pomocí funkce *generateSteps*:
  1. Provede se operace *copyAndExchange* (viz ukázka 4.8) mezi prvky sousedícími v dimenzi d.
  2. Vygenerují se stavy pro lokální seřazení na všech uzlech grafu.
  3. Dojde k redukci prvků na uzlech pomocí funkce *generateAscends* (viz ukázka 4.17).

Kapitola 5

Testování

Testování, které probíhalo během vývoje by se dalo rozdělit podobně jako implementace do dvou větších celků na testování prezentační vrstvy a testování simulace s jednotlivými algoritmy. Vývoj aplikace probíhal tak, že na *localhostu* byla uložena stránka *index.html*, která měla jediný *div* element a naimportovaný skript obsahující naší aplikaci, která se následně celá vykreslila do tohoto *divu*. Tato aplikace byla rovněž uložena na lokálním serveru a o její aktuálnost se staral *webpack* (viz Nástroje pro vývoj 3.3.1), který po každé změně kódu vygeneroval novou verzi aplikace. S pomocí nástroje *gulp* tak při každé změně došlo okamžitě k překreslení komponent v aplikaci, aniž by se musela ručně aktualizovat stránka prohlížeče. Změny v kódu se tak projevují „okamžitě“, což značně usnadňuje testování prezentační vrstvy.

5.1 Testování prezentační vrstvy

Protože prezentační vrstva má zejména vizuální charakter, probíhalo testování tímto způsobem:

1. Byla vytvořena sada testovacích dat, která měla formát popsaný v kapitole Datová reprezentace 4.1.1. Vytvořeny byly základní grafy menší velikosti, které splňovaly pouze to, že měly sadu uzlů vzájemně různým způsobem propojených hranami. Dále byla vytvořena sada prvků, jejichž identifikátory měly uzly v grafu rozděleny. Potom byla rovněž vytvořena sada kroků simulace, která měla strukturu stejnou, jak je zobrazeno v ukázce 4.3 v sekci Implementace 4.1. Tyto kroky ve skutečnosti nesimulovaly běh žádného algoritmu, jenom obsahovaly sadu po sobě jdoucích kroků, které simulovaly typické situace, které během simulace nastávají. To jsou výměny pozic prvků v jednom uzlu (na vícero uzlech zároveň), rozšíření prvků o prvky uzlu spojeného hranou atp.
2. Tato testovací data pak byla předložena naimplementovaným komponentám.

3. Implementovaly a upravovaly se funkce, které mají na starosti zobrazování průběhu simulace a zobrazování grafu.

4. Ve chvíli, kdy se simulace na testovacích datech chovala tak, jak bylo očekáváno, přešlo se na další fázi vývoje, ve které již bylo použito automatických testů.

5.2 Testování funkcí

Pro testování funkcí byla použita technologie *Jest*, která byla vytvořena společností Facebook [13] pro testování aplikací napsaných v Reactu pomocí JavaScriptu. Testovací případy jsou naspány v souboru *[název souboru].test.js*. Po spuštění příkazu *npm test* si Jest vytáhne všechny tyto soubory a spustí všechny testy v nich obsažené.

Každý test se implementuje jako volání funkce *test*, která má dva parametry. První obsahuje popis testu – tedy co test testuje. To slouží pro orientaci, jaký konkrétní test zahlásil chybu. Druhým parametrem je funkce, ve které se volají testovací funkce, jako např. *expect*, které předáme vypočítanou hodnotu a následně zavoláním metody *toEqual* určíme, čemu se má testovací případ rovnat.

5.2.1 Testování sekvenčního řazení

Implementací sekvenčních algoritmů se dá najít poměrně mnoho, dalo by se tedy předpokládat, že jsou naimplementovány správně a nebude je potřeba testovat. Nicméně, jak bylo řečeno v sekci 4.3.2, je potřeba sekvenční algoritmus pro potřeby naší aplikace upravit tak, aby průběžně ukládal všechny stavy, kterými řazení probíhá. Zároveň byl algoritmus upraven tak, aby splňoval podmínky neměnnosti (viz Ramda 3.3.2). Byla tedy napsána sada testů, která testuje, zda jsou stavy skutečně ukládány jak mají a zda je poslední stav seřazená posloupnost. Ukázka z testů je vidět v kódu 5.1.

```
1  const TestValuesNormal = [{id: 0, value: 0}, {id: 1,
       value: 1}, {id: 2, value: 2}, {id: 3, value: 3}]
2  const TestSorted = [0,1,2,3];
3  const TestInverted = [3,2,1,0];
4
5  test('selectionSort sorts already sorted', () => {
6    const sorted = selectionSort(TestSorted,
       TestValuesNormal);
7    expect(last(sorted)).toEqual([0,1,2,3]);
8  });
9
10 test('selectionSort sorts inverted array', () => {
```

```
11    const sorted = selectionSort(TestInverted,
          TestValuesNormal);
12    expect(last(sorted)).toEqual([0,1,2,3]);
13  });
14  ...
```

Kód 5.1: Testování funkce pro sekvenční řazení

5.2.2 Testování algoritmů

Každá čistá funkce, která se vyskytuje v implementaci jednotlivých algoritmů, je poměrně snadno testovatelná. Jako příklad zde uvádím testy funkcí pro generování a simulaci bitonického Mergesortu na hyperkrychli. Tyto funkce byly podrobněji popsány v implementaci 4.3.5.

```
1  test('generateNeighbours', () => {
2    expect(generateNeighbours(1, 0)).toEqual([1]);
3    expect(generateNeighbours(2, 1)).toEqual([0, 3]);
4    expect(generateNeighbours(5, 1)).toEqual([0, 3, 5,
         9, 17]);
5  });
6
7  test('generateEdges', () => {
8    expect(generateEdges(1)).toEqual([[0, 1]]);
9    expect(generateEdges(2)).toEqual([[0, 1], [0, 2], [
         1, 3], [2, 3]]);
10 });
11
12 test('generateSteps(dimension)', () => {
13   expect(generateSteps(1)).toEqual([0]);
14   expect(generateSteps(3)).toEqual([0, 1, 0, 2, 1, 0
         ]);
15   expect(generateSteps(5)).toEqual([0, 1, 0, 2, 1, 0,
          3, 2, 1, 0, 4, 3, 2, 1, 0]);
16 });
```

Kód 5.2: Testování pomocných funkcí pro hyperkrychli

Poté, co byl některý algoritmus naimplementován, pokračovalo se opět testováním prezentační vrstvy. Tím se ověřilo, že algoritmus skutečně generuje taková data, která je simulátor schopný prezentovat.

5.3 Uživatelské testování

Pro otestování aplikace byli záměrně zvoleni uživatelé, kteří nemají žádné zkušenosti s programováním a neznají prakticky žádný algoritmus. Smyslem bylo

zjistit, zda aplikaci lze používat podle předložených pokynů, aniž by uživatel dopředu mohl předpovídat, jak se má aplikace chovat. Tímto by se daly nalézt v aplikaci chyby, které by technicky zdatný uživatel nemusel objevit. Nakonec byla aplikace předložena studentovi Fakulty informačních technologií, který pro změnu objevil některé chyby, které se nedaly objevit během procházení předložených pokynů.

Během testování jsem si dělal poznámky, s čím mají uživatelé problémy. Kromě předložených pokynů jim k aplikaci nebylo nic řečeno. Pokyny byly následující:

1. Načtěte sudo-liché řazení (Even-odd sort).
2. Snižte počet prvků CPU na 1.
3. Vygenerujte graf se třemi uzly.
4. Spusťte přehrávání simulace a nechte ji doběhnout do konce.
5. Přepněte algoritmus na Shearsort.
6. Spusťte přehrávání simulace.
7. Zrychlete přehrávání a nechte doběhnout do konce.
8. Vygenerujte mřížku o velikosti 4 krát 5 se třemi prvky na každém CPU.
9. Spusťte simulaci a zkuste ji v jejím průběhu zastavit.
10. Zkuste během simulace měnit grafický styl.
11. Zastavte simulaci a vraťte se o několik kroků zpět.

5.3.1 Výstupy

Úvodní stránka

Někteří uživatelé byli zaskočeni úvodní stránkou, na které se vlastně kromě horní lišty s tlačítkem pro změnu algoritmu a stylu nic nevyskytuje. Prvním dojmem tak byla pochybnost, zda aplikace vůbec něco dělá.

Řešením by bylo na úvodní stránku umístit třeba nějaké informace o tom, k čemu aplikace slouží, nebo rovnou načíst první algoritmus v pořadí a zobrazit již vygenerovanou simulaci.

Krok č. 2

Druhým krokem v pokynech je snížit počet prvků na uzel. Některým uživatelům chvíli trvalo, než zjistili, o kterou možnost se jedná. Problém ale nejspíš způsoboval i samotný anglický jazyk, který je v aplikaci použit a na který nejsou uživatelé zvyklí. Pod pojmem „values per CPU“ tak nejspíš nevěděli, co si mají představit. Jednomu uživateli dokonce trochu déle trvalo než pochopil, kolik prvků na uzel má aktuálně nastaveno.

Krok č. 3

U tohoto kroku se uživatel poprvé dostal ke generování grafu. Zvolený algoritmus bylo Sudo-liché řazení na 1-D mřížce. Úkolem je vygenerovat graf o třech uzlech. Jeden z uživatelů hledal možnost nastavení počtu uzlů v nabídce, která se tam nikde nevyskytuje, protože se objeví až při stisku tlačítka „generate“. Nicméně nakonec tuto možnost také objevil.

Krok č. 7

Zrychlování simulace – u tohoto kroku uživatelé neměli žádné problémy. Za zmínku ovšem stojí, že všichni u zrychlování zkoušeli rychlost přepínat až na nejvyšší stupeň (aniž by o to byli požádáni) a zároveň nejvyšší stupeň, tedy desátý, uživatelům připadal až příliš rychlý a tak se všichni opět sami vrátili na nižší (devátý) stupeň, který je o poznání příjemnější na sledování. Při bližším zkoumání jsem si všiml, že simulátor skutečně zrychluje přehrávání po docela malých krocích (proto to uživatele nutilo dojít až na nejrychlejší stupeň) a zároveň je z nějakého důvodu poslední stupeň o poznání rychlejší než ten předposlední. Nejspíš by to chtělo rychlosti rozdělit rovnoměrněji a po větších krocích.

Krok č. 8

Po předchozí zkušenosti s generováním 1-D mřížky, hledali uživatelé možnost nastavení rozměrů mřížky pod tlačítkem *generate*, které namísto poskytnutí nabídky rovnou vygeneruje graf zadaných rozměrů. Tyto rozměry se ovšem u Shearsortu zadávají vlevo od možnosti změny počtu prvků na CPU. Po nalezení této možnosti pak nevěděli, které číslo označuje výšku a které šířku mřížky, neboť v aplikaci jsou rozměry označeny písmeny m a n. Řešením by tedy mohlo být označit tyto možnosti slovy *width* a *height*, ze kterých je jasnější, o které rozměry se jedná.

Ostatní

Během testování se objevily ještě následující nedostatky:

- Po doběhnutí simulace šla simulace opět spustit, ale přehrával se „prázdný“ krok, protože se simulace nacházela na konci. Řešením je na konci simulace tlačítko pro přehrání nezobrazovat, nebo na jeho místo zobrazit tlačítko pro restartování simulace, které by tu samou simulaci spustilo od začátku.
- Při krokování *zpět* byl student poněkud zmatený, jak funguje animace. Když se vrací v čase krok, který představuje redukci prvků na každém uzlu, tak dochází k posunu prvků a zároveň jejich zkopírování od sousedního uzlu. Toto potom dohromady působí docela nepřehledně. Řešením by bylo u tohoto kroku naimplementovat výjimku a namísto zkopírování prvků od sousedního uzlu zobrazit skutečný opak operace smazání prvků, tedy jejich „objevení se“.

Závěr

V rámci této práce byly představeny technologie pro tvorbu moderních webových *single-page* aplikací, jako je React a Redux, které se vztahují zejména k funkcionálnímu paradigmatu, jehož hlavní výhody byly v práci rovněž probrány. Hlavním cílem práce potom bylo vytvořit aplikaci pro webový prohlížeč, která by simulovala vybrané paralelní řadící algoritmy, jejichž řazení by zobrazovala přehlednou grafickou formou. Tohoto cíle bylo dosaženo. Byla naimplementována a otestována aplikace u které věřím, že by mohla sloužit jako učební pomůcka, která by napomáhala s pochopením naimplementovaných řadících algoritmů. Výsledná aplikace má podobu statické webové stránky s naimportovaným skriptem, takže ke svému běhu nepotřebuje žádný server.

Testování ukázalo, že aplikaci mohou uživatelé bez větších problémů používat. Po načtení aplikace lze nastavovat parametry sítě, na které lze simulovat chod vybraného algoritmu a to včetně zastavení a vracení se zpět v čase. Všechny požadavky, které byly na aplikaci kladeny, tak byly splněny.

Práce pro mě byla přínosem hlavně co se týká prohloubení mých znalostí Reactu. Dále jsem se seznámil s technologiemi D3 a postupem vykreslování grafiky do HTML5 *canvasu*. Povedlo se mi udělat další krok směrem k plně funkcionálnímu programování, na které se mi ještě nepodařilo zcela úplně přeorientovat z imperativního přístupu.

Možnosti dalšího rozšíření

Aplikace má poměrně velký potenciál směrem k další rozšiřitelnosti. Hlavně co se týká přidávání nových algoritmů, kde stačí u každého nově přidaného algoritmu nadefinovat topologii sítě a funkci, která bude simulovat samotné řazení. Jelikož bylo dopředu počítáno s využitím pouze u počítačů, nebyla aplikace implementována pro použití na mobilních zařízení, což je oblast, do které by bylo možné aplikaci také rozšířit. Jelikož je aplikace psána v Reactu, dalo by se využít například knihovny *React Native*, která by aplikaci dokázala doslova převést na mobilní aplikaci a zároveň zachovat i webovou verzi.

Dalším zajímavým rozšířením by bylo do aplikace zanést různé typy sekvenčních řazení, mezi kterými by bylo možné si volit. V rámci práce totiž bylo naimplementováno pouze Řazení výběrem.

Literatura

[1] Takada, M.: Single page apps in depth [online]. 2015, [Cited 2017-04-07]. Dostupné z: http://singlepageappbook.com/single-page.html

[2] Gonçalves, J.: Functional Programming [online]. 2015, [Cited 2017-04-07]. Dostupné z: https://medium.com/@jugoncalves/functional-programming-should-be-your-1-priority-for-2015-47dd4641d6b9

[3] Torvalds, L.: ECMAScript 2015 (ES6) and beyond [online]. 2017, [Cited 2017-05-03]. Dostupné z: https://nodejs.org/en/docs/es6/

[4] MIT: Mutability and Immutability [online]. 2015, [Cited 2017-04-07]. Dostupné z: http://web.mit.edu/6.005/www/fa15/classes/09-immutability/

[5] Coulman, R.: Thinking in Ramda [online]. 2016, [Cited 2017-04-07]. Dostupné z: http://randycoulman.com/blog/2016/05/24/thinking-in-ramda-getting-started/

[6] Sauyet, S.: Why Ramda? [online]. 2014, [Cited 2017-04-17]. Dostupné z: http://fr.umio.us/why-ramda/

[7] Mikšů, V.: Úvod do Reactu [online]. 2016, [Cited 2017-04-08]. Dostupné z: https://www.dzejes.cz/react-uvod.html

[8] de Sousa Antonio, C.: *Pro React*. New York: Springer Science+Business Media, ISBN 978-1-4842-1260-8.

[9] Danek, P.: Less is More: What Having A Single Source of Truth Really Means [online]. 2017, [Cited 2017-05-03]. Dostupné z: http://apttus.com/blog/benefits-of-a-single-source-of-truth/

[10] Marinacci, J.: HTML Canvas Deep Dive [online]. 2017, [Cited 2017-05-03]. Dostupné z: https://joshondesign.com/p/books/canvasdeepdive/chapter01.html

[11] Bostock, M.: Data-Driven Documents [online]. 2015, [Cited 2017-04-09]. Dostupné z: https://d3js.org/

[12] Jana, A.: How to integrate React and D3 [online]. 2016, [Cited 2017-04-09]. Dostupné z: http://www.adeveloperdiary.com/react-js/integrate-react-and-d3/

[13] Facebook: Jest - Painless JavaScript Testing [online]. 2017, [Cited 2017-04-23]. Dostupné z: https://facebook.github.io/jest/

Příloha **A**

Seznam použitých zkratek

CPU Central processing unit

CSS Cascading Style Sheets

DFS Depth-first search

DOM Document Object Model

GUI Graphical User Interface

HTML HyperText Markup Language

UI User Interface

Příloha B

Instalační příručka

B.1 Spuštění vývojového prostředí

Nástroje, které jsou potřeba pro běh vývojového prostředí:

- **node.js** verze 5 nebo vyšší,
- **npm** alespoň verze 3.

Dalším nástrojem je **gulp**, který lze nainstalovat pomocí *npm* – viz kód B.1.

```
1 npm install -g gulp
```

Kód B.1: Instalace gulp

Následně pokračujte podle těchto pokynů:

1. Vytvořte si adresář, do kterého chcete implementaci umístit.
2. Zkopírujte adresář se zdrojovými soubory z přiloženého cd do tohoto adresáře.
3. Nainstaluje potřebné závislosti – viz ukázka B.2.

```
1 npm install
```

Kód B.2: Instalace potřebných závislostí

Dále je potřeba mít na *localhostu* běžící server, např. *Apache*, na který je potřeba nahrát soubor *src/browser/dev.html* pod názvem *cesta-k-aplikaci/index.html*.

Před spuštěním je potřeba v souboru *src/common/config.js* změnit *urlPrefix* na cestu */cesta-k-aplikaci*, na které se nachází *index.html* na serveru.

Pro spuštění vývojového režimu použijte příkaz *gulp*. Poté, co doběhne spouštění, můžete otevřít webový prohlížeč a zadat cestu k aplikaci, která je nahrána na serveru.

B.2 Instalace aplikace

1. Upravte cestu k aplikaci v souboru *src/common/config.js* v proměnné *urlPrefix* na cestu, na které se bude aplikace nacházet na serveru.
2. Použijte příkaz *gulp build*, který vygeneruje soubory do složky *build*.
3. Všechny tyto vygenerované soubory nahrajte na server na uvedenou cestu.
4. Do stejné složky na serveru také nahrajte soubor *src/browser/dev.html* pod názvem *index.html*, ve kterém je potřeba nastavit cesty k vygenerovaným stylům a skriptům podobně, jako je zobrazeno v ukázce B.3.

```
1  <html>
2    <head>
3      <style id="stylesheet"> </style>
4      <link rel="stylesheet" type="text/css" href="app
           -82ae293b034ea572ae98.css">
5    </head>
6    <body>
7      <div id="app"></div>
8    ...
9      <script src="1-24f544de7195b273cf85.js"></script>
10     <script src="app-82ae293b034ea572ae98.js"></
           script>
11   </body>
12 </html>
```

Kód B.3: index.html

Příloha C

Uživatelská příručka

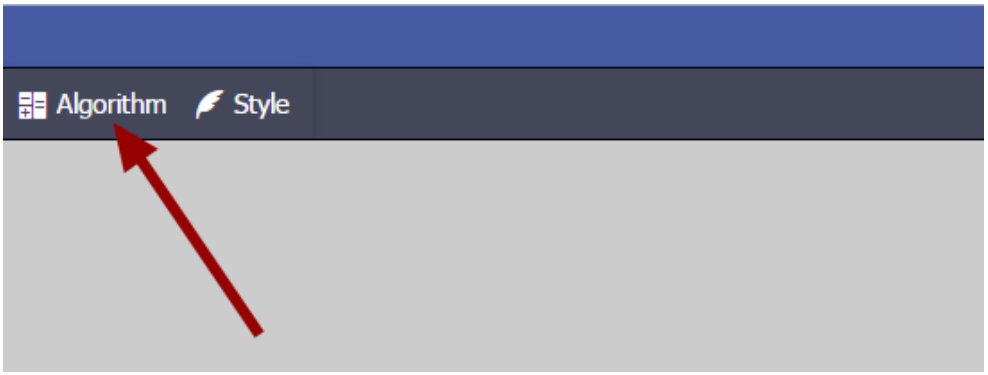
C.1 Výběr algoritmu

1. Po otevření aplikace ve webovém prohlížeči je k dispozici menu pro výběr algoritmu – viz snímek C.1.
2. Po kliknutí na tlačítko „algoritmus“ se otevře nabídka s dostupnými algoritmy.
3. Po výběru algoritmu (viz snímek C.2) se vygeneruje graf pro simulaci.

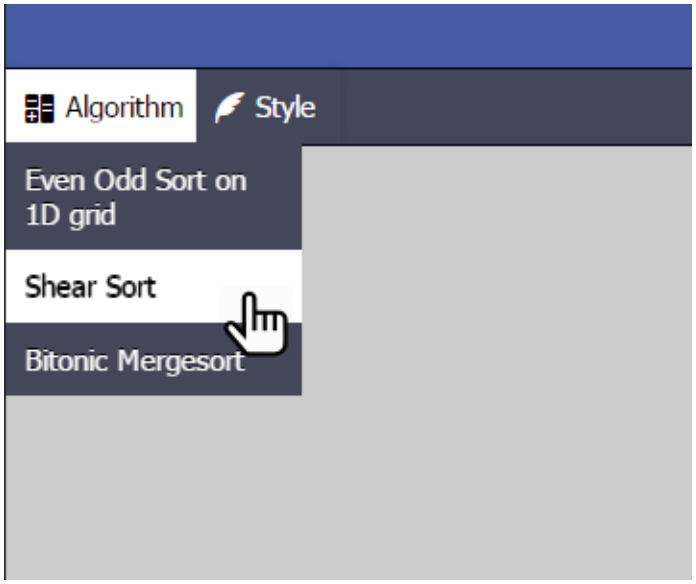
C.2 Spouštění simulace

Uprostřed spodní lišty se nachází panel pro ovládání simulace – viz snímek C.4. Tento panel obsahuje celkem tři tlačítka:

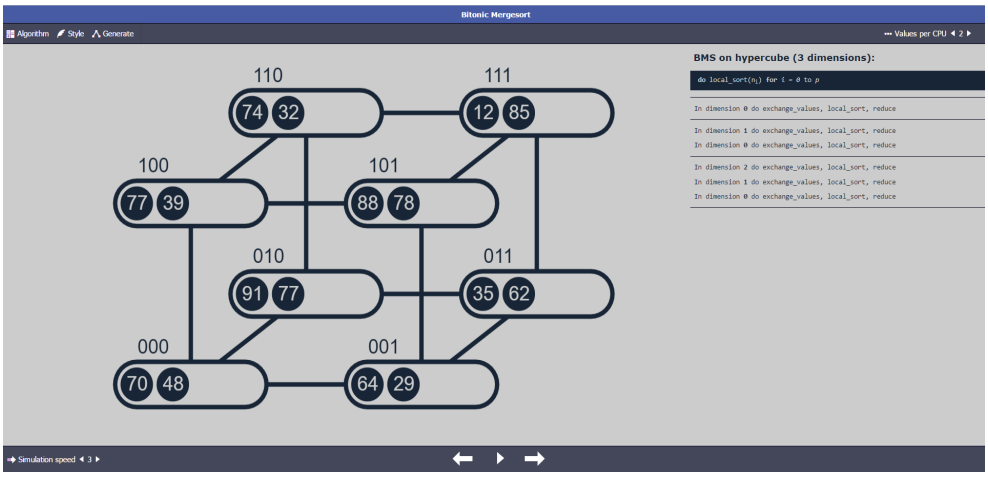
1. Pro přehrání následujícího kroku simulace, stiskněte pravou šipku.



Obrázek C.1: Snímek obrazovky - základní menu

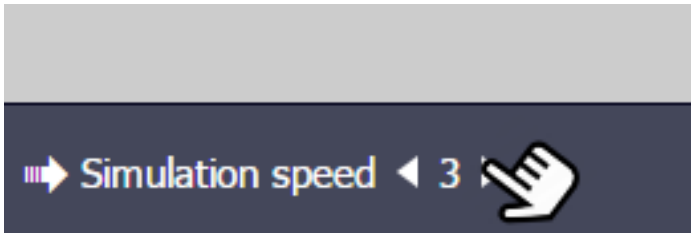


Obrázek C.2: Výběr algoritmu



Obrázek C.3: Simulátor s vygenerovaným grafem

Obrázek C.4: Panel pro ovládání simulace



Obrázek C.5: Ovládání rychlosti simulace

2. Pro navrácení k předešlému kroku simulace, stiskněte levou šipku.
3. Pro spuštění přehrávání celé simulace bez přerušení, stiskněte prostřední tlačítko „play“.

C.3 Změna parametrů simulace

C.3.1 Rychlost simulace

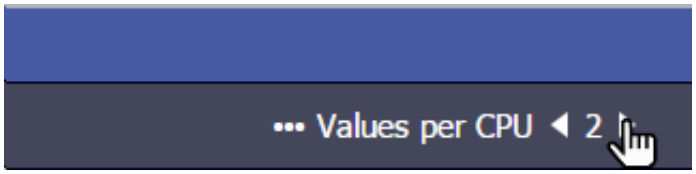
Změna rychlosti simulace je zobrazena na snímku C.5. Rychlost se mění pomocí dvou tlačítek umístěných vlevo resp. vpravo od ukazatele aktuální rychlosti. Rychlost se pohybuje v rozmezí 0 až 10.

1. Tlačítkem vlevo se aktuální rychlost simulace sníží o jeden bod.
2. Tlačítkem vpravo se aktuální rychlost simulace zvýší o jeden bod.

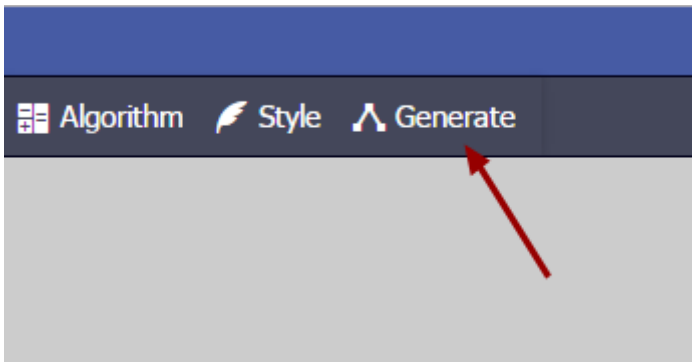
C.3.2 Počet prvků na uzel grafu

Tato možnost se nachází v pravém horním rohu aplikace.

1. Šipkou vlevo od ukazatele aktuálního počtu snížíte počet prvků na uzel o 1.
2. Šipkou vpravo od ukazatele aktuálního počtu zvýšíte počet prvků na uzel o 1.



Obrázek C.6: Počet prvků na uzel grafu



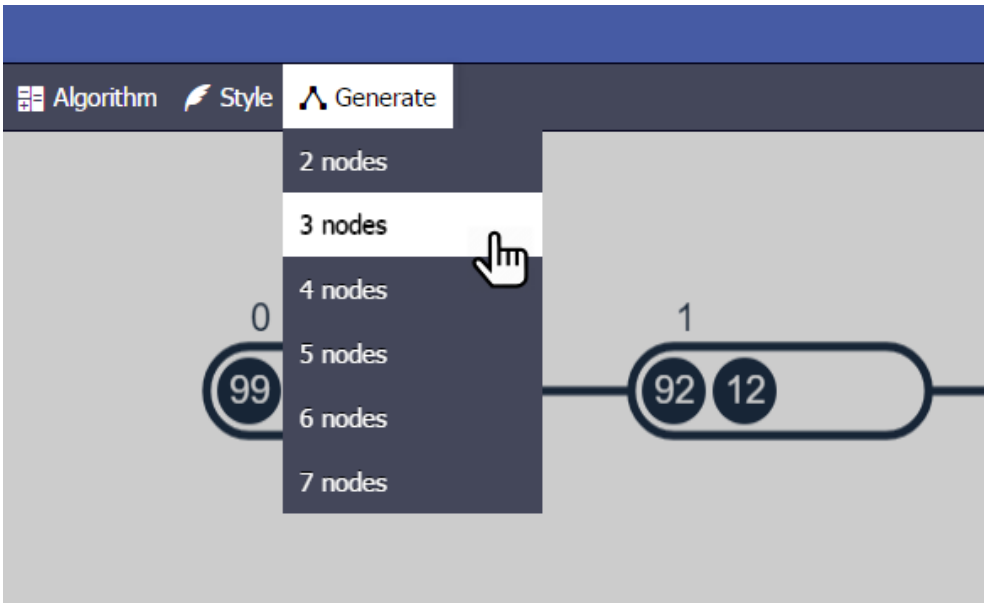
Obrázek C.7: Tlačítko pro generování simulace

C.4 Generování nové simulace

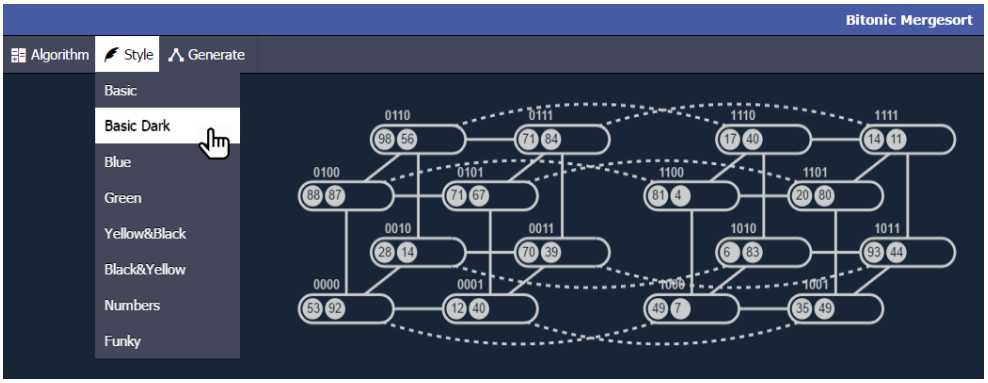
Tlačítko pro generování simulace se nachází v horní liště (viz snímek C.7). Musí již ovšem být vybrán algoritmus (viz Výběr algoritmu C.1). Po stisknutí se zobrazí lišta s nabízenými velikostmi grafu (zobrazeno na snímku C.8). Po výběru některé z možností dojde k vygenerování grafu s vybranými parametry.

C.5 Změna grafického stylu

Změna stylu se provádí přes nabídku v hlavní liště, která se zobrazí po stisknutí tlačítka „style“. Toto je zobrazeno na snímku C.9.



Obrázek C.8: Možnosti generování grafu



Obrázek C.9: Změna grafického stylu

Příloha D

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF