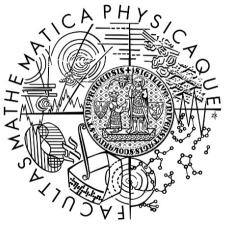


MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Duong Xuan Anh

Vizualizace a zkoušení algoritmů pro relační databáze

Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Michal Kopecký, Ph.D.

Studijní program: Informatika (B0613A140006)

Studijní obor:

IPP4 (0613RA1400060008)

Praha 2024

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V …..... dne..... podpis

Název práce: Vizualizace a zkoušení algoritmů pro relační databáze

Autor: Duong Xuan Anh

Katedra / Ústav: Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Michal Kopecký, Ph.D.

Abstrakt: Tato bakalářská práce se zaměřuje na vývoj webové aplikace, určené pro vizualizaci a interaktivní zkoušení algoritmů, potřebných pro návrh relačních databázových schémat. Primárním cílem aplikace je poskytnout studujícím možnost hlubšího porozumění a praktického procvičení návrhu schématu relační databáze, založenému na funkčních závislostech a normálních formách. Aplikace umožňuje uživatelům zadávat vlastní definici univerzálního schématu a sledovat krok za krokem, jak jednotlivé algoritmy, počínaje algoritmem pro výpočet atributového uzávěru, na zadaných datech pracují. To může uživatelům pomoci lépe porozumět teoretickým konceptům a získat zpětnou vazbu o správnosti svých řešení. Práce obsahuje rovněž teoretický základ, potřebný pro porozumění implementovaným algoritmům, což může být cenným podkladem pro studium a přípravu na zkoušky z databázových systémů.

Klíčová slova: Relační datový model, Funkční závislosti, Normální formy, Vizualizace algoritmů, Webová aplikace.

Title: Visualization of algorithms for database design

Author: Duong Xuan Anh

Department: Department of Software Engineering

Supervisor: RNDr. Michal Kopecký, Ph.D., Department of Software Engineering

Abstract: This bachelor's thesis focuses on the development of a web application designed for the visualization and interactive testing of algorithms needed for the design of relational database schemas. The primary goal of the application is to provide students with a deeper understanding and practice of relational database schema design, based on functional dependencies and normal forms. The application allows users to enter their own definition of a universal scheme and watch step-by-step how individual algorithms, starting with the algorithm for calculating the attribute closure, work on the entered data. This can help users better understand theoretical concepts and get feedback on the correctness of their solutions. The thesis also contains the theoretical basis needed for understanding the implemented algorithms, which can be a valuable basis for studying and preparing for exams on database systems.

Keywords: Relational data model, Functional dependencies, Normal forms, Algorithm visualization, Web Application.

Obsah

1 Úvod / Předmluva

Jako studentovi bakalářského programu mně (a domnívám se, že nejen mně) během studia dělalo problém plně pochopit řadu algoritmů, které jsou součástí základní přednášky z databází. Chyběla mi možnost projít si větší množství příkladů krok za krokem, zkusit si nějaké z nich vyřešit s tím, že bych měl zpětnou vazbu, zda postupuji správně, a nápovědu v případě, že při řešení udělám chybu. Cílem této práce je proto usnadnit studium dalším studentům, a navrhnout webovou aplikaci, která by umožňovala vizualizovat funkci vybraných algoritmů, důležitých pro návrh relačního schématu, a procvičování znalostí těchto algoritmů.

Podoba výsledného relačního schématu závisí na existujících vzájemných vztazích mezi jednotlivými atributy relace – funkčních závislostech – a vyžadují osvojení si celé řady algoritmů. Dekompozici, syntézu a algoritmy související – určení odvoditelnosti a redundance ve funkčních závislostech, nalezení klíčů relace, uzávěru množiny funkčních závislostí, nebo výpočet atributových uzávěrů.

Tato práce, a v ní navržená a implementovaná webová aplikace, by měla pomoci studentům zvládnout (nejen) zkouškový test z předmětu Databázové systémy (NDBI025). Tato aplikace je schopná vygenerovat vzorová řešení na základě dat, které uživatel zadá, popsat a vysvětlit jednotlivé kroky, která k řešení vedou.

Následující kapitola – Analýza – shrnuje teorii návrhu relací a jejich normalizace, potřebnou pro pochopení dané problematiky a popisuje formální algoritmy, vizualizované v navržené aplikaci. Kapitola 3 – Specifikace – na základě analýzy shrnuje základní požadavky na aplikaci. Na základě těchto požadavků jsou v kapitole 4 – Návrh aplikace – vybrány potřebné technologie. Následují kapitoly, obsahující uživatelskou příručku, návrh architektury a programátorskou příručku pro ty, kteří by chtěli aplikaci dále rozvíjet.

2 Analýza

Tato kapitola obsahuje přehled algoritmů, a potřebnou teorií, které jsou nutné vědět pro pochopení odpovídajících vizualizací. Protože podrobné zkoumání algoritmů není cílem této práce, omezíme se pouze na potřebný základ. Zde popsaná teorie vychází především ze skript [2] a slidů [3] k předmětu NDBI025 - Databázové systémy[1].

2.1 Problém návrhu databáze

Navrhnout správně strukturovanou relační databázi není triviální úkol. Jak je uvedeno v [4], normalizace relací znamená úpravu jejich struktury tak, aby relační schéma dobře reprezentovalo data, aby byla omezena redundance a přitom se neztratily vazby mezi daty. Dobře navržená relační databáze proto vyžaduje znalost teorie databázového návrhu, stejně jako porozumění aplikaci, pro kterou je databáze navrhována.

Proces návrhu můžeme rozdělit na čtyři základní fáze:

1. Analýza požadavků
2. Konceptuální návrh
3. Logický návrh
4. Fyzický návrh

V každé fázi návrhu by se měli zapojit specialisté i databázoví návrháři. Rozhodnutí, udělaná v jedné fázi mohou ovlivnit další fáze a tím i konečný design.

1. Analýza požadavků

   Dle mého názoru je tato fáze nejdůležitější, protože ostatní fáze na ní záleží. Jejím cílem je získat co nejvíce detailních informací o aplikaci prostřednictvím konzultací s klienty či specialisty tak, aby byly co nejkompletnější, co nejsrozumitelnější a konzistentní. Chyby v analýze požadavků způsobí chyby v návrhu, který pak nebude odpovídat požadavkům skutečným.

2. Konceptuální návrh

   Konceptuální návrh obsahuje zformalizování požadavků do konceptuálního modelu. Konceptuální modely umožňují modelovat data tak, aby byla

[1] https://is.cuni.cz/studium/predmety/index.php?do=predmet&kod=NDBI025

srozumitelná jak pro uživatele, tak pro vývojáře. Jako příklad můžeme uvést E-R model nebo UML model. Od studentů se očekává schopnost použít starší E-R model i novější UML model, vhodný nejen pro datové modelování.

3. Logický návrh

   Logický návrh ovlivňuje přímo organizaci dat v tabulkách, a rovněž způsob, jak se bude k datům logicky přistupovat. Logický návrh je získán převodem z konceptuálního modelu, vytvořeného v předchozím kroku. Tento převod je možné provést do značné míry algoritmicky, a studenti by měli tento lgoritmus znát a převod udělat. Výsledkem je logický datový model. V našem případě logický relační datový model, ale obdobně lze stejný konceptuální datový model převést i do jiného typu logického modelu, například objektového, nebo XML. V případě relačního modelu je možné na tomto modelu nalézt případné nežádoucí redundance a odstranit je provedením normalizace tabulek.

4. Fyzický návrh

   V této konečné fázi jsou zvoleny vhodné datové struktury. Systém řízení báze dat obvykle podporuje různé možnosti uložení datových struktur a různé přístupové metody. K určení optimálního fyzického návrhu je potřeba vědět, jak bude databáze používána uživatelem a znát relativní frekvenci uživatelských dotazů.

2.2 Návrh relačních schémat

Relace mohou být v zásadě deterministicky vytvořené v rámci kroku 3 z dříve navrženého konceptuálního modelu. Nicméně v některých případech nám může výsledek převodu vyjít neoptimálně. Když se podíváme na nějakou konkrétní relaci, která v rámci transformace na logický relační model vznikla, můžeme zjistit, že mezi jejími atributy existují nežádoucí funkční závislosti.

2.2.1 Funkční závislosti

2.2.1.1 Definice

Nechť z vlastností reálného světa vyplývá, že každé dvě entity typu R, které mají stejné hodnoty vlastností nějaké množiny X musí mít rovněž stejné hodnoty vlastností množiny Y. Na atributech A odpovídající relace R tedy musí platit stejná

závislost. Tuto závislost nazveme funkční závislostí (FZ) a budeme ji značit $f = X \rightarrow Y$, kde $X, Y \subseteq A$. Množinu všech funkčních závislostí budeme značit symbolem F.

Funkční závislost je tedy závislostí mezi dvěma množinami atributů v rámci jednoho schématu relace, a představuje dodatečné integritní omezení – vymezuje množinu přípustných relací.

2.2.1.2 **Příklad**

Mějme tabulku studijních výsledků:

Student	***Předmět***	***Známka***
Novák	Matematika	1
Novák	Fyzika	1
Horák	Biologie	3
Horák	Dějepis	3

Tabulka 1 - Funkční závislosti

Podle těchto dat to vypadá, že {Student} → {Známka}. To ale v reálném světě, který relace modeluje, obecně neplatí. Jedná se pouze o náhodu. Platnou funkční závislostí je {Student, Předmět} → {Známka}.

Existence funkčních závislostí za určitých okolností vadí, protože to vede k redundanci dat a k nežádoucím aktualizačním anomáliím. Redundance způsobí, že některou informaci máme v databázi místo jednou uloženou N-krát. To vede k tomu, že pokud děláme aktualizaci, týkající se této závislosti, může se stát, že zaktualizujeme jenom některé kopie dané informace, a jiné ponecháme v původním znění. Databáze potom nebude rozumně vypovídat o stavu reálného světa, který se snaží modelovat.

Abychom tento problém vyřešili, je možné použít různé metody a algoritmy, které budou následovně popsány, a které tyto nežádoucí funkční závislosti mezi atributy odstraní.

2.2.2 Armstrongovy axiomy

2.2.2.1 Definice

Nechť R(A) označuje relaci R se schématem A. Nechť F = { X → Y, X, Y ⊆ A } označuje množinu funkčních závislostí atributů relace R. Potom platí:

- $Y \subseteq X \subseteq A \quad \Rightarrow X \rightarrow Y$ tzv. triviální závislost, skutečný axiom
- $X \rightarrow Y \wedge Y \rightarrow Z \quad \Rightarrow Z \rightarrow Y$ tranzitivita, odvozovací pravidlo
- $X \rightarrow YZ \quad \Rightarrow X \rightarrow Y \wedge Y \rightarrow Z$ dekompozice závislosti
- $X \rightarrow Y \wedge Y \rightarrow Z \quad \Rightarrow X \rightarrow YZ$ kompozice závislostí

2.2.3 Atributový uzávěr[2]

2.2.3.1 Definice

Nechť R(A) označuje relaci R se schématem A. Nechť F = { X → Y, X, Y ⊆ A } označuje množinu funkčních závislostí atributů relace R.

Uzávěrem množiny atributů X^+ vzhledem k F je množina všech atributů funkčně závislých na X. Dále platí:

- Pokud $X^+ = A$, potom X je *nadklíčem* relace R.
- Pokud F obsahuje závislost X → Y a v X existuje atribut *a* takový, že $Y \subseteq (X - a)^+$, nazýváme *a* atributem *redundantním* v X → Y.
- FZ, která na levé straně neobsahuje žádné redundantní atributy, nazýváme *redukovanou* funkční závislostí
- FZ, která na levé straně obsahuje nějaké redundantní atributy, nazýváme *částečnou* funkční závislostí.
- Klíčem relace R je každá množina K ⊆ A taková, že je K je nadklíč (tj. platí K → A) a závislost K → A je zároveň redukovaná.
    - klíčů může existovat více, vždy minimálně jeden
    - pokud F = ∅, triviálně platí A → A, tj. klíčem je celá množina A.

2.2.3.2 Příklad

R(A, F), A = {*a*,*b*,*c*,*d*}, F = {*a* → *c*, *cd* → *b*, *ad* → *c*}

$\{a\}^+ = \{a,c\}$ kompozice triviálního *a* → *a* a *a* → *c*

$\{b\}^+ = \{b\}$ triviální *b* → *b*

$\{c\}^+ = \{c\}$ triviální *c* → *c*

$\{d\}^+ = \{d\}$ triviální *d* → *d*

[2] angl. Attribute Closure

$\{a,b\}^+ = \{a,b,c\}$ kompozice triviálního $ab \rightarrow ab$ a tranzitivního $ab \rightarrow a \rightarrow c$

$\{a,d\}^+ = \{a,b,c,d\}$ kompozice $ad \rightarrow ad$, $ad \rightarrow a \rightarrow c$ a $ad \rightarrow adc \rightarrow cd \rightarrow b$

$\{c,d\}^+ = \{b,c,d\}$ kompozice triviálního $cd \rightarrow cd$ a $cd \rightarrow b$

Odvozování atributového uzávěru pomocí Armstrongových aaxiomů je poměrně zdlouhavé. Rychlejší je využít následující algoritmus, který iterativně do uzávěru přidává atributy s využitím závislostí, vyjmenovaných v množině F.

2.2.3.3 Algoritmus

```
algorithm AttributeClosure(
                set of dependencies F,
                set of attributes X
                ): returns set X+
        ClosureX := X;
        DONE := false;
        m = |F|;
        while not DONE do
                DONE := true;
                for i := 1 to m do
                        if (LS[i] ⊆ ClosureX and RS[i] ⊄ ClosureX) then
                                ClosureX := ClosureX ∪ RS[i];
                                DONE := false;
                        endif
                endfor
        endwhile
        return ClosureX;
```

LS[i] označuje levou stranu i-té funkční závislosti a RS[i] označuje její levou stranu. Pochopit, jak se dělá atributový uzávěr je nejdůležitější základní krok v celém návrhu relačního schématu. Algoritmy pro nalezení redundantních atributů levých stran funkčních závislostí, redundantní závislosti a pro nalezení některého z klíčů relace budou zmíněné později. Všechny však využívají algoritmus *AttributeClosure* na vyhledání atributového uzávěru. Kvůli jeho důležitosti bude tento algoritmus detailně vizualizován a bude možné ho i manuálně odkrokovat.

2.2.4 Odvoditelnost funkční závislosti

Odvoditelnost funkční závislosti je klíčovým konceptem v teorii databázových systémů, který umožňuje určit, zda lze jednu funkční závislost vyjádřit nebo odvodit z jiných závislostí definovaných v rámci relačního schématu. Funkční závislost $X \rightarrow Y$ se považuje za odvoditelnou z množiny funkčních závislostí F, pokud lze pomocí pravidel odvozování, jako jsou Armstrongovy axiomy, dokázat, že Y je

funkčně závislé na X pouze s využitím závislostí uvedených v F. Prakticky se pro zjištění odvoditelnosti využívá algoritmus pro výpočet atributového uzávěru. Pokud je závislost odvoditelná, musí být všechny atributy na pravé straně funkčně závislé na atributech levé strany, tedy musí platit $Y \subseteq X^+$

2.2.4.1 **Algoritmus**

algorithm **IsDependencyInClosure**(set of dependencies F, $X \rightarrow Y$)
    return $Y \subseteq$ *AttributeClosure*(F, X);

2.2.5 Uzávěr množiny funkčních závislostí

2.2.5.1 **Definice**

Nechť R(A) označuje relaci R se schématem A. Nechť je dána množina funkčních závislostí F.

Uzávěrem F^+ množiny funkčních závislostí F je množina všech funkčních závislostí $X \rightarrow Y$ odvoditelných z F.

2.2.5.2 Příklad

Mějme relaci R(A), kde $A = \{i, j, k\}$ a $F = \{k \rightarrow j,\ ij \rightarrow k,\ k \rightarrow i\}$. Potom uzávěr množiny funkčních závislostí $F^+ = \{i \rightarrow i,\ j \rightarrow j,\ k \rightarrow ijk,\ k \rightarrow ik,\ k \rightarrow jk,\ k \rightarrow k,\ k \rightarrow ij,\ \boldsymbol{k \rightarrow i},\ \boldsymbol{k \rightarrow j},\ ij \rightarrow ijk,\ ij \rightarrow ij,\ ij \rightarrow ik,\ ij \rightarrow i,\ ij \rightarrow jk,\ ij \rightarrow j,\ \boldsymbol{ij \rightarrow k},\ ik \rightarrow ijk,\ ik \rightarrow ik,\ ik \rightarrow ij,\ ik \rightarrow i,\ ik \rightarrow jk,\ ik \rightarrow k,\ ik \rightarrow j,\ jk \rightarrow ijk,\ jk \rightarrow jk,\ jk \rightarrow ij,\ jk \rightarrow j,\ jk \rightarrow ik,\ jk \rightarrow k,\ jk \rightarrow i,\ ijk \rightarrow ijk,\ ijk \rightarrow ij,\ ijk \rightarrow ik,\ ijk \rightarrow i,\ ijk \rightarrow jk,\ ijk \rightarrow j,\ ijk \rightarrow k\}$ obsahuje celkem 37 odvoditelných funkčních závislostí.

2.2.6 Redundantní atributy

2.2.6.1 **Definice**

Atribut $a \in X$ je redundantní (nadbytečný) atribut na levé straně funkční závislosti $X \rightarrow Y$, pokud Y je funkčně závislá již na podmnožině X-*a*. Tedy, pokud FZ (X-*a*) $\rightarrow$ Y je odvoditelná z F.

2.2.6.2 Příklad

Mějme relaci R(A), kde $A = \{i, j, k, l, m\}$ a $F = \{m \rightarrow k,\ lm \rightarrow j,\ ijk \rightarrow l,\ j \rightarrow m,\ l \rightarrow i,\ l \rightarrow k\}$. Můžeme se ptát, zda jsou ve funkční závislosti $ijk \rightarrow l$ na levé straně nějaké redundantní atributy, což by znamenalo, že ve skutečnosti je to *l* určené něčím menším. Může se jednat o případ, kdy funkční závislost tvrdí, že jméno člověka je

určené rodným číslem, jeho věkem a jeho výškou, zatímco by pro určení jména stačilo i méně atributů (v tomto případě pouze rodné číslo).

Víme, že $ijk \rightarrow l$, ale můžeme se zeptat zda náhodou to l není určeno samotnými atributy jk. Tedy, můžeme se zeptat, jestli je FZ $jk \rightarrow l$ odvoditelná z F (což by znamenalo, že atribut l je funkčně závislý na dvojici atributů jk, jinými slovy že l je v atributovém uzávěru jk. S použitím algoritmu ***IsDependencyInClosure*** zjistíme, že FZ $jk \rightarrow l$ odvoditelná není, protože atributový uzávěr $jk^+ = jkm$, a neobsahuje potřebné l. Tedy i není redundantní atribut. Pokud se zeptáme, zda je k redundantním atributem, protože je odvoditelná FZ $ij \rightarrow l$. V tomto případě zjistíme, že ano, protože $ik^+ = ijmkl$.

2.2.6.3 Algoritmus

Algoritmus určuje, zda je atribut redundantní na levé straně FZ, nebo ne.

```
algorithm IsAttributeRedundant (
                set of deps. F, dep. X → Y ∈ F, attribute a ∈ X
                )
    return IsDependencyInClosure (F, X – {a} → Y);
```

Nyní si můžeme všimnout, že jsme zde nepřímo použili algoritmus ***AttributeClosure*** na vyhledání atributového uzávěru. Takže vizualizace na rozhodnutí odvoditelnosti v podstatě použije vizualizaci na určení odvoditelnosti a na vyhledání atributového uzávěru.

2.2.7 Redukce levé strany funkční závislosti

2.2.7.1 Definice

Redukovaná levá strana závislosti je taková levá strana, která neobsahuje žádný redundantní atribut.

2.2.7.2 Příklad

Mějme opět relaci R(A), kde A = $\{i, j, k, l, m\}$ a F = $\{ m \rightarrow k,\ lm \rightarrow j,\ ijk \rightarrow l,\ j \rightarrow m,\ l \rightarrow i,\ l \rightarrow k\}$. Můžeme se ptát, zda je levá strana funkční závislosti $ijk \rightarrow l$ redukované, nebo nikoli.

Algoritmus zkouší z levé strany postupně odstraňovat jednotlivé atributy a kontroluje, zda je získaná FZ odvoditelná z F. Pokud ne, je atribut potřebný – neredundantní, a je potřeba ho zachovat. Pokud ano, je naopak redundantní, a je možné ho z levé strany odstranit.

Z předchozího příkladu víme, že atribut *i* je neredundantní a nemůžeme jej odstranit. Dále můžeme zjistit, zda je redundantní atribut *j*. S použitím algoritmu ***IsDependencyInClosure*** zjistíme, že FZ $ik \rightarrow l$ odvoditelná není, protože atributový uzávěr $ik^+ = ik$. Tedy ani *i* možné z levé strany odstranit. Z předchozího příkladu již víme, že FZ $ij \rightarrow l$ odvoditelná je, a atribut *k* je tedy redundantní a může být z levé strany odstraněn. Formálně lze tento postup zapsat následujícím algoritmem:

2.2.7.3 Algoritmus

Algoritmus, který vrátí zredukovanou levou stranu FZ.

algorithm **GetReducedAttributes**(set of deps. F, dep. $X \rightarrow Y \in F^+$)
    X' := X;
    for each a ∈ X **do**
        if *IsAttributeRedundant*(F, X' → Y, a) **then** X' := X' – {a};
    endfor
return X';

Opět si můžeme všimnout, že jsme nepřímo (několikrát) použili algoritmus ***AttributeClosure*** na vyhledání atributového uzávěru..

Je nutno podotknout, že výsledek hledání neredundantní levé strany FZ není jednoznačný, a závisí na pořadí, ve kterém se pokoušíme atributy odstraňovat. Proto by měl program dovolit uživateli měnit pořadí atributů v definici funkčních závislostí. Tato funkce mi u ostatních programů velmi chyběla. Pokud program dovolí pouze závislost přepsat, je to časově neefektivní a uživatel může snadno udělat při přepisování chybu.

2.2.8 Redundantní závislosti

2.2.8.1 Definice

Redundantní závislosti jsou nadbytečné závislosti, odvoditelné z ostatních, které proto chceme z definice F odstranit.

2.2.8.2 Příklad

Pokud nám někdo popisuje svět, který máme modelovat, většinou v popisu použije řadu zbytečných (redundantních) informací ohledně funkčních závislostí. Například můžeme pro entitní typ R se čtyřmi atributy a jí odpovídající relaci R(A), A = {a, b, c, d} obdržet množinu funkčních závislostí:

$$F = \{a \rightarrow c,\ b \rightarrow a,\ b \rightarrow c,\ d \rightarrow a,\ d \rightarrow b,\ d \rightarrow c\}$$

U každé jednotlivé závislosti f ∈ F můžeme zjistit, zda ji lze odvodit ze zbývajících závislostí. Pokud bychom například uvažovali první závislost f = $a \rightarrow c$, dokázali bychom jí odvodit ze zbylých závislostí? Tedy, nachází se závislost f v uzávěru množiny závislostí F-f? Problém je ekvivalentní tomu, zda je pravá strana závislosti obsažená v atributovém uzávěru levé strany, tedy zda $c \in a^+$ vzhledem k množině závislostí F-f. S použitím algoritmu ***AttributeClosure*** zjistíme, že $a^+ = a$. Závislost $a \rightarrow c$ tedy není odvoditelná ze zbývajících, není redundantní, a odstranit ji nemůžeme. Pokud se pokusíme postup zopakovat pro závislost $b \rightarrow c$, zjistíme, že $b^+ = bc$ vzhledem k F-{$b \rightarrow c$}. Závislost $b \rightarrow c$ tedy je odvoditelná ze zbývajících, je redundantní, a odstranit ji můžeme.

Formálně můžeme tento algoritmus popsat následovně:

2.2.8.3 Algoritmus

algorithm **IsDependencyRedundant** (
set of dependencies F, dependency X→Y ∈ F
)
return IsDependencyInClosure(F – {X → Y}, X → Y);

Opět platí, že algoritmus prostřednictvím algoritmu ***IsDependencyInClosure*** nepřímo používá algoritmus ***AttributeClosure***. To znamená, že na zjištění redundantních závislostí je potřeba použít algoritmus ***AttributeClosure*** na vyhledání atributového uzávěru.

Protože algoritmus ***IsDependencyRedundant*** funguje podstatě stejně jako ***AttributeClosure***, pro který předpokládáme vizualizaci, a navíc ho používáme, abychom našli minimální pokrytí, nemusí být bezpodmínečně vizualizovaný, ale bude případně stačit pouze textové vysvětlení.

Je nutno podotknout, že výsledek opět není jednoznačný, a může záviset na pořadí, ve kterém se pokoušíme závislosti odstraňovat. Nejednoznačnosti, kdy existuje více "stejně dobrých" výsledků komplikuje pochopení těchto algoritmů.

2.2.9 Minimální pokrytí

Pochopení výše uvedených algoritmů je potřebné pro získání tzv. minimálního pokrytí množiny funkčních závislostí.

2.2.9.1 Definice

Nechť je zadaná relace R(A) se schématem A a množinou funkčních závislostí F. Potom množina závislostí G je minimálním pokrytím množiny F, pokud:

- G *je pokrytím* množiny F, tedy $F^+ = G^+$.
  Tedy množina všech odvoditelných závislostí je shodná.
- G je kanonické.
  Tedy obsahuje pouze elementární funkční závislosti s jedním atributem na pravé straně.
- G obsahuje pouze redukované funkční závislosti.
  Žádná levá strana tedy neobsahuje žádné redundantní atributy.
- G je neredundantní.
  Tedy neobsahuje žádnou redundantní závislost.

Konstruuje se odstraněním nejprve redundantních atributů v závislostech a až potom redundantních závislostí. Cílem je, zmenšit popis funkčních závislostí co nejvíce, a odstranit z něj všechny informace, které lze odvodit ze zbytku. Postup je velmi podobný tomu, když je z levé strany závislosti potřeba odstranit redundantní atributy.

2.2.9.2 Příklad

Mějme relaci R(A,F), kde A = {*a*, *b*, *c*, *d*, *e*} a F = {$abcd \rightarrow e$, $e \rightarrow d$, $a \rightarrow b$, $ac \rightarrow d$}

Celý algoritmus má 3 kroky.

Prvním krokem je rozepsání všech závislostí na závislosti elementární. Toho lze snadno dosáhnout využitím pravidla o dekompozici funkční závislosti v rámci Armstrongových axiomů.

Druhým krokem je odstranění redundantních atributů na levých stranách funkčních závislostí. V našem příkladu jsou atributy *b* a *d* redundantní ve funkční závislosti $abcd \rightarrow e$.

Třetím krokem je odstranění redundantních závislostí. Aplikací algoritmu odstranění redundantních závislostí na množinu F' = {$ac \rightarrow e$, $e \rightarrow d$, $a \rightarrow b$, $ac \rightarrow d$}, zjistíme, že funkční závislost $ac \rightarrow d$ je redundantní a lze ji tedy odvodit ze zbývajících závislostí.

Výsledkem aplikace tohoto postupu na množinu funkčních závislostí bude tedy minimální pokrytí G = {$ac \rightarrow e$, $e \rightarrow d$, $a \rightarrow b$}

2.2.9.3 **Algoritmus**

algorithm **GetMinimumCover**(
set of dependencies F
): returns minimal cover G
decompose each dependency in F into elementary ones;
for each X → Y **in** F **do**
F := (F – {X → Y}) ∪ {*GetReducedAttributes*(F, X → Y) → Y};
endfor
for each X → Y in F **do**
if *IsDependencyRedundant*(F, X → Y) **then** F := F – {X → Y};
endfor
return F;

Získání minimálního pokrytí je tedy kombinací několika výše uvedených algoritmů – redukce levých stran a odstranění redundantních funkčních závislostí. Znamená to, že pro vizualizaci minimálního pokrytí je potřeba v zásadě využít vizualizaci výpočtu atributového uzávěru, nalezení redundantních atributů a nalezení redundantních závislostí.

2.2.10 Klíče

2.2.10.1 **Definice**

Nadklíčem relace R(A,F) je každá (na čase nezávislá) podmnožina atributů K ⊆ A taková, že hodnoty v těchto sloupcích společně jednoznačně identifikují každou řádku, tedy že FZ K → A je odvoditelná z F. Pokud zároveň platí, množina K tvořících nadklíč, je vzhledem k inkluzi minimální taková, tedy že FZ K → A je redukovaná, je K *klíčem* relace R.

Relace může mít více klíčů. Ten který vybereme jako rozhodující, nazýváme primárním klíčem. Platí, že kombinace všech atributů A relace R(A, F) určitě jednoznačně určuje každou n-tici, protože FZ A → A je triviální závislostí, ale nemusí být vzhledem k inkluzi minimální. Redukcí levé strany FZ A → A vždy dostaneme nějaký klíč relace. V extrémním případě je klíčem celé A.

2.2.10.2 **Příklad**

Mějme 5 prvkovou relaci R(A, F), kde A = {*a*, *b*, *c*, *d*, *e*} a F = {*a* → *c, ae* → *d, d* → *a, a* → *c, c* → *b*}.

Nyní odstraňme všechny redundantní atributy na levé straně v triviální funkční závislosti *abcde* → *abcde*. Pravá i levá strana se skládá ze všech atributů ve schématu A. Na pořadí odstraňování atributů záleží (různá pořadí mohou najít různé

redukované levé strany, tedy klíče), ale při libovolně zvoleném pořadí se vždy jeden z klíčů najde.

V prvním případě zkusme postupovat od *a* do *e*, tedy zleva do prava. Zeptáme se jestli i bez atributu *a* budou na zbývajících atributech funkčně závislé všechny atributy *abcde*. To zjistíme algoritmem ***AttributeClosure***. $bcde^+ = abcde$. Atribut *a* je tedy možné odstranit. Pokračujeme stejně a vyjde nám, že i atributy *b* a *c* je možné odstranit. Atributy *d* a *e* odstranit nejdou. Ve výsledku dostaneme $de^+ = abcde$, tedy $de \rightarrow abcde$, a tedy *de* je jeden z klíčů.

Zkusme odstraňovat atributy v opačném pořadí, tedy zprava doleva.

Stejným způsobem zjistíme, že *e* nelze odstranit, takže ho tam necháme. *d*, *c* a *b* jsou redundantní a odstranit jdou. Nakonec *a* není redundantní, proto ho odstranit nemůžeme. Ve výsledku dostaneme $ae^+ = abcde$, tedy $ae \rightarrow abcde$, což znamená, že *ae* je rovněž klíčem relace.

2.2.10.3 Algoritmus vyhledání prvního klíče

algorithm **GetFirstKey**(set of deps. F, set of attributes A) : **returns a key** K;
    return *GetReducedAttributes*(F, A → A);

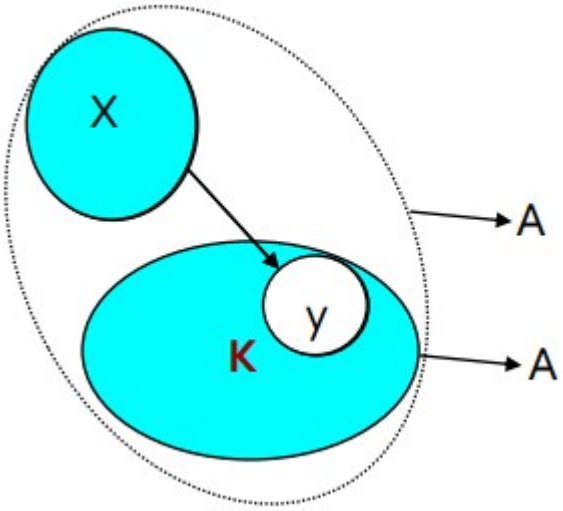
Vyhledávání prvního klíče je aplikací algoritmu ***GetReducedAttributes*** na FZ A → A. To znamená, že vizualizace vyhledání prvního klíče je stejná jako vizualizace tohoto algoritmu.

2.2.10.4 Algoritmus vyhledání všech klíčů

Mějme schéma R(A, F), kde F je ve formě minimální pokrytí s elementárními závislostmi.

1. Vezměme jeden libovolný klíč K, nalezený algoritmem ***GetFirstKey***
2. Vezměme některý z již nalezených klíčů K a funkční závislost $X \rightarrow y \in F$ takovou, že $y \in K$ a $X \not\subseteq K$. Pokud neexistuje, tak konec (není žádný jiný klíč).
3. Protože $X \rightarrow y$ a $K \rightarrow A$, tranzitivně také platí $X \cup \{K - y\} \rightarrow A$, což znamená, že $X \cup \{K - y\}$ je nadklíčem relace R.
4. Redukcí levé strany funkční závislost $X \cup \{K - y\} \rightarrow A$ dostaneme klíč K' na levé straně. Tento klíč je určitě jiný než klíč K (odstranili jsme y).
5. Jestliže K' není mezi již nalezenými klíči, přidáme ho mezi nalezené klíče.

6. Opakujme od bodu 2, dokud se neotestují všechny kombinace nalezených klíčů a vhodných funkčních závislostí.



Obrázek 1 - Odvození dalšího (nad)klíče [3]

Formálně lze tento algoritmus zapsat takto:

algorithm **GetAllKeys**(
set of deps. F, set of attributes A
) : **returns set of all keys** Keys;
let all dependencies in F be non-trivial,
i.e. replace every X → Y by X → (Y – X)
FIFO Q;
N := *GetFirstKey*(F, A);
Keys := {N};
PUSH(Q, N);
while not empty(Q) **do**
K := POP(Q);
for each X → Y ∈ F **do**
if (Y ∩ K ≠ Ø **and** ∄K' ∈ Keys : K' ⊆ (K ∪ X) –Y) **then**
N := *GetReducedAttributes*(F, ((K ∪ X) –Y) → A);
Keys := Keys ∪ {N};
PUSH(Q, N);
endfor;
endwhile;
return Keys;

2.2.11 Normalizace

Dle [1] a [6] ovlivňuje struktura tabulek (jejich počet, v nich obsažené atributy a jejich vzájemné funkční závislosti) množství možných datových anomálií a tedy kvalitu celkového řešení. Proces rozhodování o celkové struktuře tabulek nazýváme datovým modelováním. V rámci tohoto procesu je často – vzhledem k nežádoucím

funkčním závislostem – potřeba provést optimalizaci návrhu vzhledem k informační potřebě i potřebám rozumné implementace. Spolu s teorii relačních DB byla vypracována technologie takové optimalizace, která se nazývá normalizace.

Normalizace relací znamená úpravu jejich struktury tak, aby relační schéma dobře reprezentovalo data, aby byla omezena redundance a přitom se neztratily vazby mezi daty. Autorem je pan E.F. Codd, který definoval tři úrovně normalizace. Je to formalizovaný postup, kterým se postupně zlepšuje struktura relací tak, aby se minimalizoval výskyt nežádoucích anomálií v datech. Uplatněním pravidel, které navrhl pan Codd se upřesňuje struktura dat pro daný DB od 0NF přes 1NF, 2NF, 3NF ke konečnému řešení.

Výhody normalizace:

- vede k lepšímu uložení dat v databázi.
- omezuje redundanci.
- snižuje riziko nekonzistence.

Nevýhody normalizace:

- zvyšuje četnost operace spojení, která je paměťově i časově náročná, proto někdy raději volíme nenormalizované relace kvůli rychlejší odezvě.
- může značně znepřehlednit databázi, protože struktura tabulek neodpovídá struktuře entit reálného světa.

2.2.11.1 **První normální forma (1NF)**

- První normální forma (1NF) v databázovém modelování je základním stavebním kamenem normalizace. Aby tabulka splňovala 1NF, musí mít všechny atributy (sloupce) atomické hodnoty, tedy hodnoty, které nejsou dále dělitelné. To znamená, že každý atribut obsahuje jedinou hodnotu z definovaného doménového oboru, a v tabulce se nevyskytují žádné opakující se skupiny dat.

Příklad: [3]

- Osoba(Id: Integer, Jméno: String, Příjmení: String, Narozen: Date) je v 1NF
- Zaměstnanec(Id: Integer, Podřízení: Osoba[], Nadřízený: Osoba) je 0NF, ale není v 1NF, která nedovoluje atribut strukturovaného typu pole prvků typu (Podřízení), ani atribut strukturovaného typu (Nadřízený).

Další příklad (převzato z [1] a upraveno, strana 47 až 52):

V relaci, obsahující data o osobách, bychom chtěli mít u každé osoby možnost pamatovat si více telefonních čísel. Dostali bychom následující tabulku Osoba:

ID	***Jméno***	***Příjmení***	***Adresa***	***Telefon***
1	David	Duong	Praha	789465132, 6547912348
2	Petr	Novák	Ústí nad Labem	124567946, 456798654
3	Maria	Nováková	Brno	456789321, 321132465

Tabulka 2 - Tabulka, nesplňující 1NF

S takovouto tabulkou by byla spojena řada problémů, například by se dost špatně prováděly změny telefonních čísel a případné vyhledávání podle telefonního čísla by bylo možné jen komplikovaně. Abychom dostali relační schéma, splňující 1NF, musíme oddělit telefonní čísla do samostatné tabulky. Výsledné schéma bude vypadat následovně:

Tabulka Osoba:

ID	***Jméno***	***Příjmení***	***Adresa***
1	David	Duong	Praha
2	Petr	Novák	Ústí nad Labem
3	Maria	Nováková	Brno

Tabulka Telefon:

ID_osoby	***Číslo***
1	789465132
1	6547912348
2	124567946
2	456798654
3	456789321
3	321132465

Tabulka 3 - Relační schéma, splňující 1NF

Dodržení 1NF je pro správnou funkci relační DB v podstatě povinné. Je naštěstí většinou dodržována intuitivně.

2.2.11.2 **Druhá normální forma (2NF)**

Druhá normální forma klade následující podmínky:

- tabulka musí být v první normální formě (1NF)
- každý neklíčový atribut musí být plně závislý na každém kandidátním klíči (neklíčovým atributem rozumíme atribut, který není součástí žádného kandidátního klíče)

Druhá normální forma klade důraz především na odstranění možných duplicit v záznamech.

Příklad: (převzato z [3] a upraveno)

Mějme tabulku DbFirma(*Firma*, *DB server*, *Sídlo*, *Rok zakoupení*), kde F = {Firma, DB Server → Rok zakoupení, Firma → Sídlo}, a jediným klíčem je tedy dvojice atributů Firma, DB Server:

Firma	***DB server***	***Sídlo***	***Rok zakoupení***
A	Oracle	Brno	2010
A	MS SQL	Brno	2015
B	Oracle	Praha	2020
B	IBM SQL	Praha	1999
B	MS SQL	Praha	2004

Tabulka 4 - Tabulka, nesplňující 2NF

Tato tabulka není v 2NF, protože neklíčový atribut *Sídlo* závisí již na části klíče *Firma*. Abychom tuto tabulku převedli do 2NF opět musíme tabulku rozdělit, a to následovně:

Tabulka DbFirma:

Firma	***DB server***	***Rok zakoupení***
A	Oracle	2010
A	MS SQL	2015
B	Oracle	2020
B	IBM SQL	1999
B	MS SQL	2004

Tabulka SídloFirmy:

Firma	***Sídlo***
A	Brno
B	Praha

Tabulka 5 - Relační schéma, splňující 2NF

2.2.11.3 **Třetí normální forma (3NF)**

Třetí normální forma klade následující podmínky:

- tabulka je ve druhé normální formě (2NF)
- tabulka neobsahuje tranzitivní závislosti neklíčových atributů na jakémkoli kandidátním klíči.

Příklad: [3] Mějme tabulku Firma(*Firma*, *Sídlo*, *PSČ*), kde F = {*Firma* → *PSČ*, *PSČ* → *Sídlo*}, a tedy (jediným) klíčem relace je atribut Firma:

Firma	***Sídlo***	*PSČ*
A	Praha	14000
B	Brno	22012
C	Praha	10000
D	Brno	22012
E	Ústí nad Labem	41301

Tabulka 6 - Tabulka, nesplňující 3NF

Tato tabulka je ve 2NF, ale není ve 3NF (tranzitivní závislost Sídla na klíči přes PSČ). Nežádoucím důsledkem je redundance hodnot atributu *Sídlo*.

Abychom získali relační schéma ve 3NF, opět musíme tabulku rozdělit, a to následovně:

Tabulka Firma, kde F = {*Firma* → *PSČ*}:

Firma	***PSČ***
A	14000
B	22012
C	10000
D	22012
E	41301

Tabulka PSČ, kde F = {*PSČ* → *Sídlo*}:

PSČ	***Sídlo***
14000	Praha
22012	Brno
10000	Praha
41301	Ústí nad Labem

Tabulka 7 - Relační schéma, splňující 3NF

2.2.11.4 **Boyce–Coddova normální forma (BCNF)**

Relace R(A,F) je v BCNF je-li v 1NF a pro každou množinu atributů $C \subseteq A$ a každý atribut $a \notin C$ platí: $C \rightarrow a \Rightarrow C \rightarrow A$. Jinými slovy, každá netriviální závislost atributu je závislostí na (nad)klíči relace. Pokud je relace v BCNF, potom je i v 3NF. Naopak tvrzení neplatí. Podmínka pro BCNF je silnější a vyplývá z ní, že relace už neobsahuje žádné tranzitivní závislosti neklíčových atributů na klíči. Významným důsledkem je, že aktualizace hodnoty nějaké n-tice ovlivní pouze tu n-tici, která byla aktualizována.

Při řešení příkladů poznáte BCNF tak, že levý atribut implikuje pouze atributy které jsou na levé straně a nebo atributy na levé straně jsou klíčem všech atributů na pravé straně.

Příklad: [3]

Mějme tabulku Lety(*Destinace*, *Pilot*, *Letadlo*, *Den*), kde F = {*Pilot*, *Den* → *Destinace*, *Letadlo*, *Letadlo*, *Den* → *Destinace*, *Destinace* → *Pilot*}, a klíči relace jsou dvojice atributů {*Pilot*, *Den*} a {*Letadlo*, *Den*}:

Destinace	***Pilot***	***Letadlo***	***Den***
Praha	kpt. Novák	Boeing 777	sobota
Praha	kpt. Novák	Boeing 777	pátek
Londýn	kpt. Horák	Airbus A320	sobota

Tabulka 8 - Tabulka, nesplňující BCNF

Tato tabulka je ve 3NF, ale není v BCNF (atribut *Pilot* závisí na atributu *Destinaci*, což není nadklíč). Důsledkem je redundance hodnot atributu *Pilot*.

Abychom se dostali do BCNF tak opravíme tabulku takto:

Tabulka Destinace, kde F = {*Destinace* → *Pilot*}:

Destinace	***Pilot***
Praha	kpt. Novák
Londýn	kpt. Horák

Tabulka Lety, kde F = {*Letadlo*, *Den* → *Destinace*}:

Destinace	***Letadlo***	***Den***
Praha	Boeing 777	sobota
Praha	Boeing 777	pátek
Londýn	Airbus A320	sobota

Tabulka 9 - Relační schéma, splňující BCNF

2.2.11.5 **Shrnutí normalizace**

Při řešení příkladů je nejjednodušším způsobem, jak určit správně normální formu postup od BCNF směrem k 2NF. Pokud není relace v BCNF může být v 3NF. Pokud ne, může být v 2NF. Nakonec, pokud není v 2NF je v obvykle předpokládané 1NF.

2.2.12 Dekompozice

Tento algoritmus rozkládá jednu větší relaci (nazývanou jako univerzální relace) na několik jejích projekcí. Je založen na opakovaném rozkladu jedné z relací R relačního schématu v nevyhovující NF na dvojici menších relací R_1 a R_2, přičemž každý krok zaručuje tzv. bezztrátové spojení, tedy že $R_1 * R_2 = R$.

2.2.12.1 **Algoritmus**

Vstup: Schéma relace R(A,F).

Výstup: Relační schéma databáze **R** = $\{R_i(A_i, F_i), 1 \leq i \leq n\}$

declare
RESULT je množinová proměnná obsahující po skončení algoritmu R,
DONE: Boolean.
begin
RESULT := {R};
DONE := FALSE;
Vytvoř F^+; // F^+ je množina všech závislostí, které lze odvodit z F v daném schématu R(A). (Může jich být až $(2^{|A|} - 1)^2$)
while (**not** DONE) **do**
if v RESULT existuje R_i, které není v BCNF
then begin
nechť $X \rightarrow Y \in F^+$ je netriviální funkční závislost,
nechť $X \rightarrow A_i \notin F^+$;
RESULT := (RESULT - $R_i(A_i)$)
$\cup$ R_i (A_i - Y, $F^+ \upharpoonright$ (A_i - Y))
$\cup$ R_j(X $\cup$ Y, $F^+ \upharpoonright$ (X $\cup$ Y))
end
else DONE := TRUE;
end

Je důležité zdůraznit, že výsledné schéma je v BCNF a dekompozice splňuje vlastnost bezztrátového spojení. Nicméně, nemusí být zaručeno, že bude zachována vlastnost pokrytí závislostí. To znamená, že pro výsledek bude platit, že $(\cup(F_i))^+ \subset F^+$, a nikoli $(\cup(F_i))^+ = F^+$. Algoritmus má rovněž tu nevýhodu, že je nutné explicitně vypočítat uzávěr F^+. Pro vzniklá schémata je dále nutné určit všechny klíče, aby bylo možné zjistit jejich NF.

2.2.12.2 **Příklady**

Příklad 1 [2, str. 64-66]:

Mějme následující tabulku Výuka, kde (po zkrácení názvů atributů počátečními písmeny) A = (*P*, *U*, *M*, *H*, *S*, *Z*) a F = {$P \rightarrow U$; $H,M \rightarrow$ P; $H,U \rightarrow M$; $P,S \rightarrow$ Z; $H,S \rightarrow M$}:

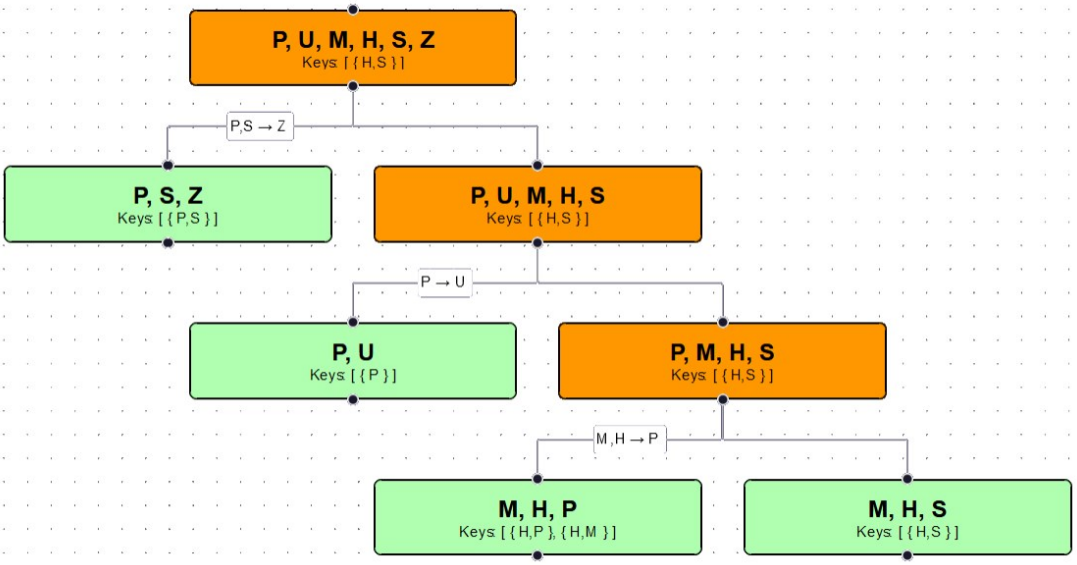
Přednáška	***Učitel***	***Místnost***	***Hodina***	***Student***	***Známka***
Programování	Kryl	S7	Po9	Novák	2
Programování	Kryl	S3	Út3	Novák	2
Programování	Kryl	S7	Po9	Volák	3

Přednáška	*Učitel*	*Místnost*	*Hodina*	*Student*	*Známka*
Programování	Kryl	S3	Út3	Volák	3
Systémy	Král	S4	Po7	Zíka	1
Systémy	Král	S4	Po7	Tupý	2
Systémy	Král	S4	Po7	Novák	1
Systémy	Král	S4	Po7	Bílý	2

Tabulka 10 - Tabulka, nesplňující 3NF

V tomto jednoduchém případě můžeme provést dekompozici do BCNF (nebo 3NF) hledáním tranzitivních závislostí, které narušují požadovanou normální formu. Výsledná schémata budou dostatečně jednoduchá, aby testování na splnění BCNF nebylo příliš obtížné.

Dekompozici budeme reprezentovat binárním stromem, kde jedna ze dvou hran vycházejících z dekomponovaného schématu bude označena funkční závislostí, podle které se provádí dekompozice.



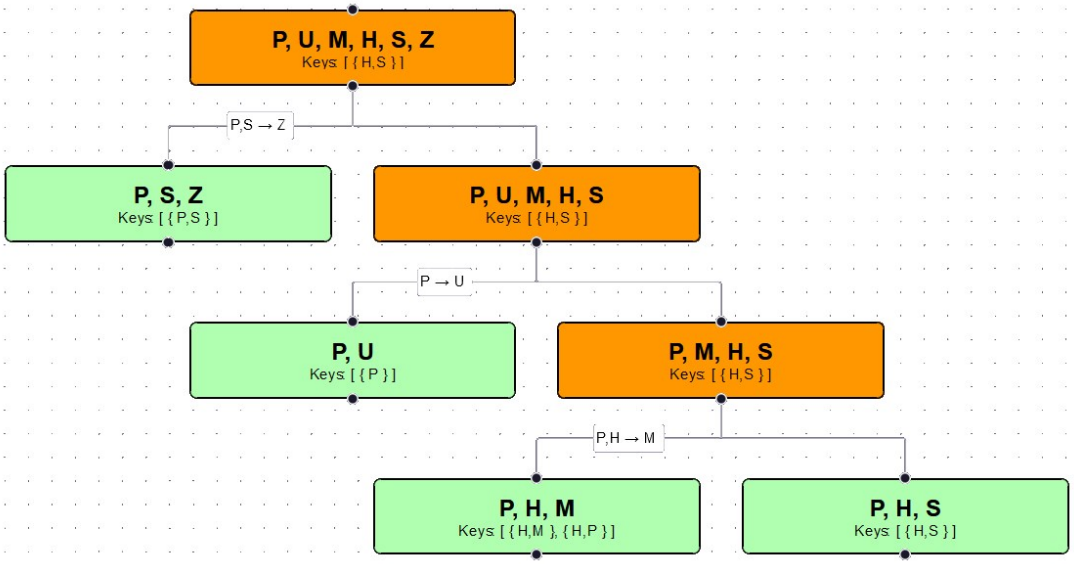
Obrázek 2 - Dekompozice do BCNF - varianta 1

Postupem, uvedeným na obrázku výše, obdržíme dekompozici $R_{III} = \{R_1(\{\underline{P,S},Z\}, \{P,S \rightarrow Z\}), R_2(\{\underline{P},U\}, \{P \rightarrow U\}), R_3(\{\underline{M,H},P\}, \{M,H \rightarrow P, P,H \rightarrow M\}), R_4(\{M,\underline{H,S}\}, \{H,S \rightarrow M\})\}$

Poznámka:

Schéma MHP má dva klíče – dvojice atributů *P*,*H* a *M*,*H*.

Jiná varianta dekompozice téhož schématu vypadá následovně (Klíče schématu jsou podtrženy):



Obrázek 3 - Dekompozice do BCNF - varianta 2

Výsledné schéma je tedy:

$R_{IV} = \{R_1(\{P,S,Z\}, \{P,S \rightarrow Z\}), R_2(\{P,U\}, \{P \rightarrow U\}), R_3(\{P,H,M\}, \{P,H \rightarrow M\}), R_4(\{P,H,S\}, \{H,S \rightarrow P\})\}$

Pokud bychom v posledním kroku zkusili dekomponovat podle závislosti $H,S \rightarrow M$, (algoritmus dekompozice tuto možnost neuvažuje, protože H,S je klíčem relace, ale výsledek vyhovuje požadavku na bezztrátovost), obdrželi bychom výsledek

$R_{II} = \{R_1(\{P,S,Z\}, \{P,S \rightarrow Z\}), R_2(\{P,U\}, \{P \rightarrow U\}), R_3(\{H,S,M\}, \{H,S \rightarrow M\}), R_4(\{P,H,S\}, \{H,S \rightarrow P\})\}$.

Je důležité si uvědomit, že všechna tři výsledná relační schémata jsou z hlediska algoritmu dekompozice ekvivalentní. Avšak z hlediska užitečnosti bychom možná upřednostnili schéma R_{IV}, protože sledování toho, ve které místnosti se právě nacházejí studenti, není pro uživatele databáze důležité. Dekompozice by tedy měla být procesem návrhu schématu databáze, který řídí uživatel.

Je pravda, že dekompozice nemusí zachovat pokrytí závislostí, což se projevuje i v tomto příkladě. Například v schématu R_{II} ztrácíme závislosti $H,M \rightarrow P$ a $P,H \rightarrow M$.

Původní tabulka 8, pokud by nerespektovala tyto funkční závislosti, by mohla obsahovat například následující řádky:

Přednáška	*Učitel*	*Místnost*	*Hodina*	*Student*	*Známka*
Programování	Valenta	S7	St3	Sochor	2
DBS	Valenta	S7	St3	Bílý	3
DBS	Valenta	S4	St3	Nová	2

Tabulka 11 - Data, nerespektující ztracené funkční závislosti

Tedy, v jednu hodinu (St3) se v jedné místnosti (S7) učí dva různé předměty, a jeden předmět (DBS) se v jednu hodinu (St3) učí ve dvou místnostech. V databázi budou vytvořeny čtyři relace:

Přednáška	***Student***	*Známka*
Programování	Sochor	2
DBS	Bílý	3
DBS	Nová	2

Přednáška	*Učitel*
Programování	Valenta
DBS	Valenta

Místnost	***Hodina***	***Student***
S7	St3	Sochor
S7	St3	Bílý
S4	St3	Nová

Přednáška	***Hodina***	***Student***
Programování	St3	Sochor
DBS	St3	Bílý
DBS	St3	Nová

Tabulka 12 - Relační schéma R_{II}

Příklad 2:

Mějme relační schéma R(A,F), kde A = (a, b, c, d) a F = {$a \rightarrow b$, $b \rightarrow c$, $c \rightarrow d$}

Pokud v prvním kroku dekompozice rozdělíme R podle FZ $b \rightarrow c$, dostaneme relace R_1(b, c) a R_2(a, b, d), což je z hlediska algoritmu v pořádku. Ztratíme ovšem FZ $b \rightarrow c$. V F_2 tedy musíme použít odvozenou závislost $b \rightarrow c \in F^+$. Následně dokážeme určit, že klíčem R_2 je atribut a, a tedy R_2 není v BCNF ani v 3NF kvůli tranzitivní závislosti $a \rightarrow b \rightarrow d$.

2.2.13 Syntéza

Syntéza – v rozporu se svým názvem – rovněž rozděluje jednu tabulku s mnoha atributy v nevyhovující normální formě na více tabulek s méně atributy v dostatečně vysoké normální formě.

2.2.13.1 **Algoritmus**

Algoritmus syntézy vychází za zadané množiny atributů a funkčních závislostí podobně jako u dekompozice.

Vstup: schématu relace R(A,F), kde A je množina atributů a F je množina funkčních závislostí.

Výstup: **R** = {$R_i(A_i, F_i)$, $1 \leq i \leq n$}.

1. Vytvořit minimální pokrytí G množiny FZ F.
2. Závislosti z G se roztřídí do skupin, přičemž v každé skupině jsou závislosti se stejnou levou stranou. Atributy závislostí každé skupiny tvoří schéma jedné relace, vzniklé syntézou. Atributy levé strany zároveň tvoří klíč (jeden z klíčů) této relace.
3. Jsou-li v takto vzniklých relacích takové, kde schéma jedné relace je podmnožinou schématu druhé relace, lze menší relaci z výsledku odstranit.
4. Jsou-li v takto vzniklých relacích takové, které mají shodné, nebo funkčně ekvivalentní klíče, je vhodné zvážit jejich sloučení v jediné schéma. Důvodem je, že obě schémata popisují jeden objekt. Mohlo by se ovšem stát, že ve sloučeném schématu budou platit nežádoucí funkční závislosti, a výsledná normální forma bude příliš nízká (méně než 3NF).

Výsledek kroků (1) až (3) bude alespoň ve 3NF a z kroku (1) plyne, že budou zachovány všechny závislosti.

Pro zachování bezztrátovosti spojení je potřeba, aby alespoň jedna ze vzniklých relací obsahovala alespoň jeden univerzální klíč původní relace. Pokud tomu tak není, je potřeba za krok (4) vložit krok

5. Nechť K je jeden z univerzálních klíčů relace R(A,F). Přidej do výsledného relačního schématu **R** relaci $R_{n+1}(K, \varnothing)$.

2.2.13.2 **Příklad**

Mějme relaci Contracts(A,F), kde A = {c = ContractId, s = SupplierId, j = ProjectId, d = DeptId, p = PartId, q = Quantity, v = Value} a F = {$c \rightarrow s,j,d,p,q,v$; $s,d \rightarrow p$; $p \rightarrow d$; $j,p \rightarrow c$; $j \rightarrow s$} [3, slide 22][3].

Prvním krokem je najít minimální pokrytí. Po použití výše uvedeného algoritmu dostaneme následující výsledek minimálního pokrytí.

$$G = \{c \rightarrow j, c \rightarrow p, c \rightarrow q, c \rightarrow v, sd \rightarrow p, p \rightarrow d, jp \rightarrow c, j \rightarrow s\}$$

Po úpravě dostaneme:

$$G' = \{c \rightarrow jpqv, sd \rightarrow p, p \rightarrow d, jp \rightarrow c, j \rightarrow s\}$$

Z minimálního pokrytí vytvoříme relace. Za každou funkční závislost vznikne jedna relace. To znamená, že obdržíme následující relační schéma

$R_1(\underline{c, j, p}, q, v)$, $F_1 = \{c \rightarrow jpqv, jp \rightarrow c\}$ … BCNF,

$R_2(\underline{s, d}, p)$, $F_2 = \{sd \rightarrow p, p \rightarrow d\}$ … 3NF, pozn: druhým klíčem je klíč $\{s, p\}$

$R_3(\underline{p}, d\}$, $F_3 = \{p \rightarrow d\}$ … BCNF,

$R_4(\underline{j, p, c}\}$, $F_4 = \{jp \rightarrow c, c \rightarrow jp\}$ … BCNF,

$R_5(\underline{j}, s\}$, $F_5 = \{j \rightarrow s\}$ … BCNF

Atributy na levé straně příslušné funkční závislosti tvoří jeden z klíčů vzniklé relace. Klíče jsou označené podtržením. Vidíme, že relace R_3 je podmnožinou relace R_2, takže ji můžeme z výsledku odstranit. Obdobně vidíme, že relace R_4 je podmnožinou relace R_1.

Po odstranění těchto relací obdržíme relační schéma:

$R_{1,4}(\underline{c, j, p}, q, v)$, $F_1 = \{c \rightarrow jpqv, jp \rightarrow c\}$ … BCNF,

$R_{2,3}(\underline{s, d}, p)$, $F_2 = \{sd \rightarrow p, p \rightarrow d\}$ … 3NF, pozn: druhým klíčem je klíč $\{s, p\}$

$R_5(\underline{j}, s)$, $F_5 = \{j \rightarrow s\}$ … BCNF

[3] https://www.ms.mff.cuni.cz/~kopecky/vyuka/dbs/lecture09.pdf, citováno 13. 2. 2024

Dalším vhodným krokem je sloučení dvojic relací, jestliže mají shodné, nebo funkčně ekvivalentní klíče. Při slučování je ovšem třeba dávat pozor na to, že výsledná relace může obsahovat nežádoucí funkční závislosti, které brání dosažení alespoň třetí normální formy.

V tomto příkladu jsou klíče $\{c\}$, $\{j, p\}$ a $\{j, d\}$ ekvivalentní, protože v původní relaci Contracts funkčně závisí jeden z druhém. Jedná se o klíče této relace. Ke sloučení relací R_1 a R_4 došlo již v předchozím kroku.

Metoda syntézy zaručuje pokrytí funkčních závislostí, ale nezaručuje bezztrátovost spojení. Abychom bezztrátovosti dosáhli, je potřeba, aby alespoň jeden jeden z původních klíčů univerzální relace (zde máme klíče $\{c\}$, $\{j, p\}$ a $\{j, d\}$) je obsažený alespoň v jedné výsledné relaci. V našem případě je klíč $\{c\}$ (a rovněž klíč $\{j, p\}$) obsažen v relaci $R_{1,4}$, a syntéza je tedy bezztrátová. V opačném případě by bylo potřeba přidat některou z relací $R_6(\underline{c})$, $R_6(j, p)$, nebo $R_6(j, d)$, kde $F_6 = \varnothing$.

2.2.14 Existující programy

2.2.14.1 Bakalářský projekt Dany Soukupové

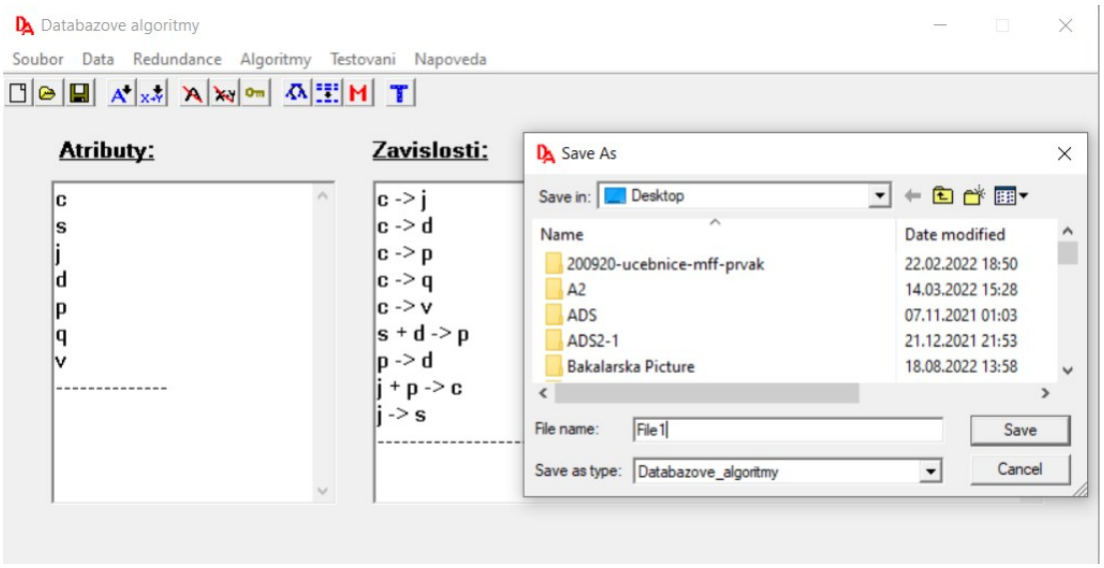
Program, který mi během studia předmětu Databázové systémy (NBI025) velmi pomohl, je bakalářská práce *Algoritmy logického návrhu relační databáze* od Dany Soukupové[4] [5].

Výhody programu:

+ Dokáže určit redundantní atributy, redundantní závislost, klíče, dekompozice, syntézu a maximální množinu. (Všechny tyto pojmy již máme definované předchozích bodech).
+ Možnosti ukládání použitých dat.
+ Umožňuje uživateli si vyzkoušet příklady. Tuto funkci jsem používal velmi často a hodně mi pomohla s kontrolou řešení úlohy.

Všechny výhody v programu od Dany Soukupové proto chci mít implementované i v mém programu.

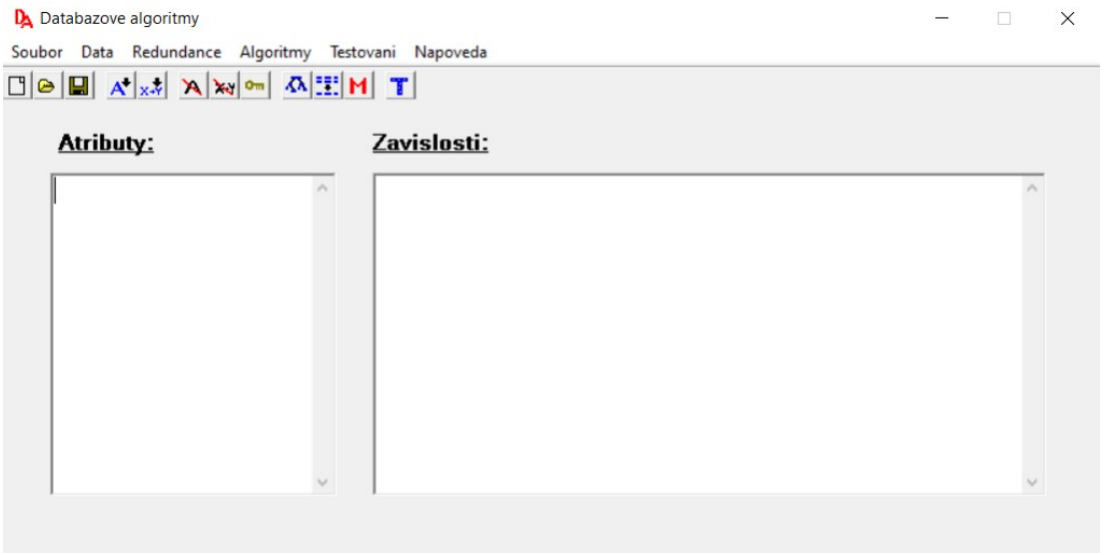
[4]https://www.ms.mff.cuni.cz/~kopecky/vyuka/ndbi025/siret.ms.mff.cuni.cz/skopal/databaze/DatAlg.zip



Obrázek 4 - Okno pro ukládání dat [5]

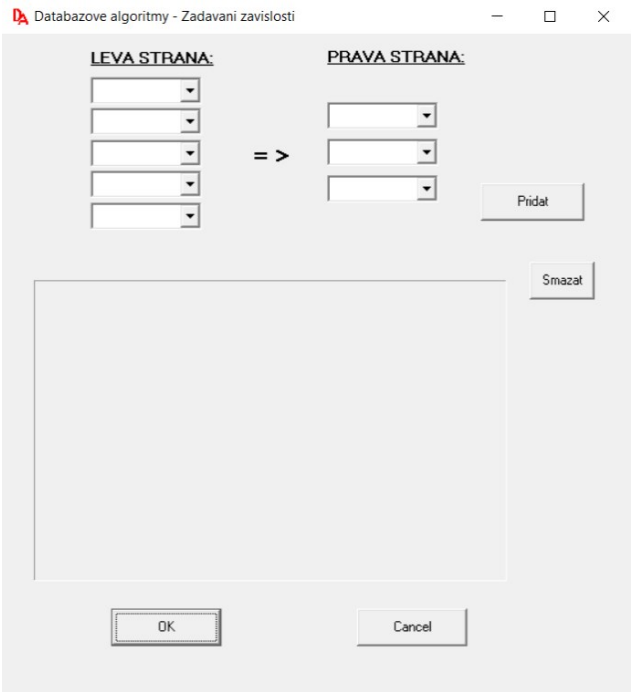
Nevýhody programu:

- Program je nutné instalovat lokálně do počítače s operačním systémem MS Windows.
- Program bohužel nefunguje zcela korektně, a v některých případech vypočítá špatně minimální pokrytí. Například pro výše uvedený příklad Contracts(c, s, j, d, p, q, v} a F = {$c \rightarrow s$,j,d,p,q,v; s,$d \rightarrow p$; $p \rightarrow d$; j,$p \rightarrow c$; $j \rightarrow s$} vrací výsledek {$c \rightarrow j$; $c \rightarrow q$; $c \rightarrow v$; s,$d \rightarrow p$; $p \rightarrow d$; j,$p \rightarrow c$, $j \rightarrow s$}. Tento výsledek není správný. Ve výsledku by měla být navíc závislost $c \rightarrow p$, případně $c \rightarrow d$, v závislosti na tom, v jakém pořadí by se jejich redundance testovala.
- Zastaralý vzhled.
- Při zadávání funkčních závislostí má program omezení na počet atributů na pravé i levé straně.
- Program dostatečně nevysvětluje postup dekompozice, což může začátečníkovi působit obtíže.
- Dekompozice nerozlišuje 1.NF a 2.NF, takže uživatel neví jaká je skutečná normální forma dané relace.
- Zobrazuje pouze výsledek, nikoliv postup či vysvětlení.
- Ikony nejsou příliš intuitivní.



Obrázek 5 - Design - hlavní okno aplikace

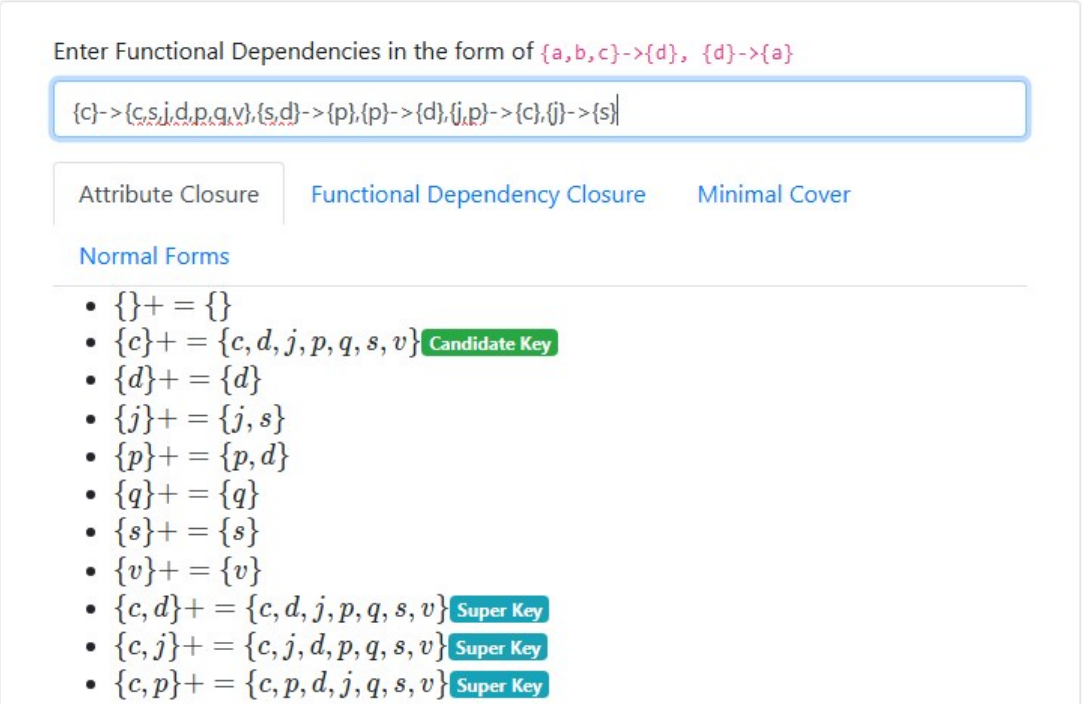
Vzhled aplikace pochází z doby Windows XP. Při pohledu na použité ikony je nesnadné určit jejich funkcionalitu. Vzhled okna budí dojem, že je možné atributy a závislosti zapisovat přímo do textových polí, zatímco je potřeba je přidávat je s pomocí nástrojové lišty, a pole slouží jen pro jejich zobrazování.



Obrázek 6 - Okno pro zadávání atributů relace

Program má vestavěná omezení na počet zadávaných atributů. Na levé straně program přijímá maximálně 5 atributů a na pravé straně maximálně 3 atributy. Dle mé zkušenosti, to nemusí být dostačující pro zadávání složitější univerzální relace, které jsem běžně jako student potkával, nebo se v praxi může požadovat.

2.2.14.2 Functional Dependencies Checker[5]



Obrázek 7 - Functional Dependencies Checker

Jedná se o webovou aplikaci, která nám na základě funkčních závislostí dokáže určit:

- Klíče
- Atributový uzávěr (Attribute closure)
- Uzávěr množiny funkčních závislostí F^+ (Functional dependency closure)
- Minimální pokrytí (Minimal cover)
- Normální formy (Normal forms)

Výhody programu:

\+ Všechno určuje rychle a správně.

Nevýhody programu:

\- Nevysvětluje výsledek.

[5] https://arjo129.github.io/functionalDependencyCalculator/

2.2.14.3 Normalization Tool od Griffith university[6]

Tento nástroj je výsledkem projektu v oblasti vzdělávání a výuky, financovaného Griffith University, na kterém spolupracovali docent Junhu Wang a profesor Bela Stantic. Uživatelské rozhraní vytvořil Dr. Xuguang Ren.

Je navržen tak, aby pomohl studentům získat znalosti o funkčních závislostech, normálních formách a normalizaci. Lze jej použít pro testování zadaného relačního schématu z hlediska normálních forem nebo pro její normalizaci do 2NF, 3NF nebo BCNF na základě dané sady funkčních závislostí.

Výhody programu:

+ Program se vyznačuje atraktivním designem a je schopen identifikovat a vysvětlovat minimální pokrytí, kandidátní klíče a normální formy. Navíc umožňuje normalizaci tabulek až do požadované normální formy.

Nevýhody programu:

- Program čelí problémům s dostupností, kdy je často přetížený a nedostupný. Dále je vysvětlování výsledků nedostatečně přehledné, což může uživatele vést k potížím s orientací ve výstupech.

2.2.14.4 Algovision or how to animate algorithms od Luděk Kučera

Pro vizualizace algoritmů existuje webová aplikace Algovision[7] od Luďka Kučery. Program byl vyvinut na Karlově univerzitě jako podpora pro studenty kurzu *Algorithm and Data Structures*. Jako student jsem Algovision občas používal na pochopení algoritmů a datových struktur v programování a myslím si, že vizualizace mi velmi usnadnila práci. Program ale nepodporuje vizualizace databázových algoritmů, což mě velmi mrzelo. A to je důvodem, proč jsem se rozhodl vyvinout webovou aplikaci pro vizualizaci databázových algoritmů pro studenty předmětu Databázové systémy - NDBI025 na Univerzitě Karlově.

2.2.15 Shrnutí analýzy

Vizualizace funkčních závislostí při studiu předmětu *Databázové systémy* (NDBI025) na Univerzitě Karlově mi pro pochopení jednotlivých postupů velmi

[6] http://www.ict.griffith.edu.au/normalization_tools/normalization/

[7] https://www.algovision.org/

chyběla. Pokud by uživatel chtěl pouze výsledek, bylo by vhodné dovolit vizualizaci přeskočit.

Algoritmy předchozích kapitol jsou popsané formálně, aby bylo zřejmé, jak fungují. Ve skutečnosti je potřeba algoritmy často rozepsat detailněji a rozdělené do více fází, tak aby podporovaly krokování a vizualizaci co nejlépe.

3 Specifikace

Z úvodních požadavků, a následné podrobnější analýzy vyplývá, že pokud má navrhovaná aplikace předčit ty stávající a již dostupné, a v co největší míře pomoci studentům s pochopením probírané látky, měla by splňovat řadu požadavků.

3.1 Typ aplikace

Mělo by se jednat o aplikaci, zaměřenou na vizualizaci a interaktivní zkoušení algoritmů pro návrh relačních databázových schémat, nejen na výpočet výsledku a jeho zobrazení.

3.2 Responzivita

Aby bylo možné aplikaci použít na různých zařízeních od klasického desktopového počítače přes notebooky, a tablety až po mobilní telefony, aplikace by měla být plně responzivní.

3.3 Uživatelské rozhraní

Aplikace by na všech typech klientských zařízení měla nabídnout prostředky pro:

- *Zadávání vstupních dat*: Uživatel by měl mít pohodlnou možnost definovat univerzální schéma relace s atributy a funkčními závislostmi pro následnou analýzu a procvičování.
- *Vizualizace algoritmů*: Jednotlivé algoritmy by mělo být možné vizualizovat a vysvětlovat pomocí textového popisu, a kde to bude výhodné či potřebné, tak s využitím grafického znázornění.
- *Krokování a animace*: V případě potřeby nebo zájmu by mělo být možné procházet algoritmy krok za krokem.
- *Integraci interaktivního průvodce*: Aplikace by měla poskytovat možnost přímo v aplikaci vysvětlovat logiku a účel každého kroku algoritmu.

Vzhledem k nezanedbatelnému počtu zahraničních studentů by uživatelské rozhraní mělo být k dispozici jak v českém, tak anglickém jazyce s možností pozdějšího rozšíření o podporu dalších jazyků.

3.4 Přístupnost

Aplikace by měla být navržena tak, aby byla přístupná a snadno použitelná pro všechny uživatele. Uživatelé by měli mít možnost použít program kdykoliv, kdekoliv

je k dispozici připojení k internetu bez nutnosti cokoliv instalovat a bez ohledu na operační systém, který používají.

3.5 Rozšiřitelnost a udržitelnost

Aplikace by měla mít modulární design, aby umožnila snadné rozšíření a přidání nových funkcí bez narušení stávající funkcionality.

Pro programátora by měla být k dispozici dostatečná programátorská dokumentace, která by mu snadnou údržbu a zamýšlené rozšíření usnadnila.

4 Návrh aplikace

4.1 Technologie

Aby byla aplikace snadno přístupná, je aplikace navržená jako webová. Z těchto důvodů byly vybrány následující technologie, aby v co největší míře usnadnily projekt realizovat.

4.1.1 JavaScript

JavaScript je nenáročný, interpretovaný nebo právě včas kompilovaný jazyk s prvotřídními funkcemi (chovají se jako proměnné). Přesto, že je JavaScript známý především jako jazyk, používaný na klientské straně ve webovém prohlížeči, je dne velmi široce používán i mimo toto prostředí. Například Node.js[8], použitý i pro vývoj této aplikace, je určen pro vývoj serverové části aplikací. Je použitý například i v Apache CouchDB. JavaScript je prototypově založený[9], jedno-vláknový, dynamický jazyk, který podporuje objektové programování a funkční styl programování.

4.1.2 React

React[10] je populární frontendová knihovna pro tvorbu efektivních uživatelských rozhraní. Základním kamenem Reactu jsou jeho komponenty, tedy znovupoužitelné části kódu. Komponenty si mohou mezi sebou předávat data skrze svá properties. Novinkou Reactu jsou tzv. hooky, díky kterým je jednodušší psát znovu-použitelnou logiku do funkcí, které mohou používat všechny ostatní komponenty. React komponenty poté renderují blok JSX. JSX je XML/HTML podobná syntaxe, používaná Reactem. Umožňuje tak použití kódu napsaného JavaScriptu/Reactu uvnitř HTML struktury. JSX je poté přeloženo preprocesorem (například Babel[11]) do JS kódu, který poté procedurálně vytváří jednotlivé HTML elementy a k nim příslušný JS kód.

[8] https://nodejs.org/
[9] tj. třídy nejsou explicitně definované, ale pouze jsou postupně rozšiřované o další vlastnosti.
[10] https://react.dev/
[11] https://babeljs.io/

Existuje mnoho dalších knihoven, jako například Vue[12] nebo Angular[13]. Každá z nich má svoje výhody a nevýhody. Ty ale nehrají až tak zásadní roli – záleží především na tom, v čem se člověku pracuje nejlépe.

Dalším důvodem pro výběr knihovny React pro můj projekt byla její funkcionalita React Context. React Context[14] je mechanismus, který umožňuje efektivní sdílení dat mezi komponentami bez nutnosti použití techniky zvané prop drilling. Prop drilling vyžaduje předávání dat skrze props přes více úrovní komponent, což může vést k nadměrné složitosti a snížené přehlednosti kódu. Inicializace kontextu probíhá pomocí funkce React.createContext(), což umožňuje následné vytvoření Provider komponenty. Tato komponenta slouží jako zdroj dat a distribuuje je mezi vnořené komponenty, které tato data vyžadují. Použití React Contextu mi umožnilo udržet kód aplikace čistý a minimalizovat redundanci ve spojení komponent, což zásadně zjednodušilo správu stavu a dat v aplikaci. Tento přístup byl klíčový pro efektivní řízení stavu v mé aplikaci a výrazně přispěl k modularitě a udržitelnosti celého projektu.

4.1.3 Zefektivnění navigace v React aplikacích s použitím React Router DOM

React Router DOM[15] je pokročilá knihovna pro routování v aplikacích vyvíjených pomocí Reactu. Tato knihovna umožňuje vývojářům implementovat komplexní navigační struktury s využitím dynamického routování. Díky tomu je možné efektivně načítat pouze ty části uživatelského rozhraní, které jsou v dané chvíli potřebné, což zlepšuje rychlost a odezvu aplikace.

Významnou výhodou React Router DOM je jeho schopnost podporovat vnořené a podmíněné routy, díky čemuž můžete snadno spravovat složité uživatelské scénáře a závislosti mezi komponentami. To je zvláště užitečné pro aplikace, které vyžadují řízení přístupu na základě uživatelských oprávnění nebo pro dynamické zobrazování obsahu v závislosti na stavech aplikace.

Ve svém projektu jsem využil React Router DOM k vytvoření čisté a uživatelsky přívětivé navigace mezi různými částmi aplikace. Integrace této knihovny umožňuje

[12] https://vuejs.org/
[13] https://angular.io/
[14] https://legacy.reactjs.org/docs/context.html
[15] https://reactrouter.com/en/main

uživatelům plynule přecházet mezi stránkami bez zbytečného zatěžování serveru, což vede k rychlejšímu a efektivnějšímu provozu aplikace.

4.1.4 Optimalizace kaskádových stylů pomocí SASS

SASS[16] představuje preprocesor CSS, který značně zjednodušuje a zefektivňuje proces tvorby stylů webových stránek. Tento nástroj umožňuje používání proměnných, vnořených pravidel, mixínů a funkcí, které přinášejí do procesu designu větší flexibilitu a modularitu. SASS podporuje vývojáře v psaní čistšího, udržitelnějšího a snadněji spravovatelného kódu díky funkcím, které překračují běžné možnosti čistého CSS.

Významnou vlastností SASS je jeho schopnost dělení kódu do více souborů bez dopadu na výkon stránky, což umožňuje lepší organizaci stylů a jejich snadnější údržbu. Vývojáři mohou definovat styl jednou a znovu jej použít v celém projektu, což minimalizuje redundanci kódu a zvyšuje efektivitu celého vývojového procesu.

Ve vytvářené aplikaci je SASS využito k vytvoření konzistentního a responzivního designu uživatelského rozhraní. Použití proměnných a mixínů umožnilo rychle reagovat na změny designu bez nutnosti přepisování CSS kód pro každý jednotlivý prvek. Díky tomu byla stylizace aplikace nejen efektivnější, ale i přehlednější a snadno rozšiřitelná.

4.1.5 Firebase: Integrovaná platforma pro vývoj aplikací

Firebase[17], platforma pro vývoj aplikací od společnosti Google, poskytuje širokou škálu nástrojů a služeb, které jsou nezbytné pro efektivní vývoj, testování a optimalizaci aplikací. Jedním z hlavních důvodů pro rozhodnutí využít Firebase v projektu je její schopnost integrovat různé vývojové nástroje do jednotného ekosystému, což výrazně zjednodušuje správu projektu.

Klíčovou službou, kterou Firebase nabízí, je NoSQL databáze Cloud Firestore. Tato databáze umožňuje real-time synchronizaci dat mezi klientem a serverem, což je ideální pro aplikace, vyžadující vysokou míru interaktivity a aktuálnosti dat. V projektu je Cloud Firestore využit pro ukládání a správu uživatelských dat. Tato technologie umožnila rychle implementovat robustní a škálovatelné řešení pro správu dat bez nutnosti složité instalace a konfigurace databázové vrstvy.

[16] https://sass-lang.com/
[17] https://firebase.google.com/

4.1.6 i18next a React-i18next

i18next[18] představuje framework pro lokalizaci, který je navržen tak, aby umožňoval snadné přidávání jazykových překladů do aplikací. Díky své flexibilitě a širokému spektru podporovaných funkcí se i18next stává ideálním řešením pro projekty, které cílí na globální publikum. React-i18next, integrace i18next pro React, dále zjednodušuje správu vícejazyčných aplikací, vytvářených v Reactu. Tato knihovna poskytuje vývojářům efektivní nástroj pro implementaci lokalizace, která je přizpůsobitelná a snadno spravovatelná přímo z komponent React.

V projektu je React-i18next právě z důvodu jeho výborné integrace s Reactem, což umožnilo efektivně spravovat jazykové zdroje a usnadnilo lokalizaci uživatelského rozhraní. Tato technologie nejenže podporuje dynamické načítání překladů, ale také umožňuje snadnou změnu jazyků za běhu aplikace. To přispívá k výraznému zlepšení uživatelské zkušenosti.

4.1.7 Vizualizace dat s využitím Dagre a React Flow Renderer

Dagre[19] je knihovna pro automatické uspořádání grafů, která je optimalizovaná pro vizualizaci složitých struktur, jako jsou organizační schémata nebo různé typy diagramů. Tato knihovna poskytuje algoritmy, které umožňují efektivní rozvržení uzlů a hran tak, aby byl graf co nejpřehlednější a nejčitelnější.

React Flow Renderer[20] je robustní knihovna pro React, která se specializuje na renderování interaktivních grafů. Díky komponentově orientované povaze Reactu umožňuje tato knihovna snadno integrovat interaktivní grafy do moderních webových aplikací. React Flow Renderer podporuje funkce jako je táhni a pusť (drag and drop), zoomování a mnoho dalších interaktivních prvků, které zvyšují uživatelskou interaktivitu a zlepšují celkovou uživatelskou zkušenost.

V projektu jsou obě tyto technologie zkombinovány pro vytvoření dynamických a interaktivních grafických reprezentací dat. Dagre je využit k automatickému uspořádání grafů na základě vstupních dat, zatímco React Flow Renderer umožňuje tyto grafy efektivně zobrazovat a poskytnout uživateli možnost s nimi interaktivně pracovat.

[18] https://www.i18next.com/
[19] https://github.com/dagrejs/dagre
[20] https://reactflow.dev/

4.1.8 Zvýšení interaktivity uživatelského rozhraní pomocí React Beautiful DND

React Beautiful DND[21] je knihovna, která poskytuje vysoce optimalizované a snadno použitelné řešení pro implementaci "drag-and-drop" (táhni a pusť) funkcionality v aplikacích vytvořených pomocí Reactu. Tato knihovna umožňuje vývojářům bezproblémově přidávat interaktivní prvky do uživatelských rozhraní, což zahrnuje úkoly jako řazení seznamů, přesouvání položek mezi panely nebo konfiguraci dashboardů.

Jedním z hlavních přínosů React Beautiful DND je její schopnost zvládat složité interakce s uživatelským rozhraním s vynikající výkonností a přístupností. Knihovna poskytuje bohaté API, které umožňuje detailní nastavení chování táhni a pusť operací, včetně možnosti přizpůsobení vizuální zpětné vazby během přetahování.

V projektu je React Beautiful DND využit k vytvoření dynamických seznamů, kde uživatelé mohou intuitivně organizovat položky – například atributy relačního schématu nebo funkční závislosti – pomocí drag-and-drop interakcí. Tato funkcionalita nejenže zlepšila uživatelskou zkušenost tím, že umožnila uživatelům přizpůsobit si rozhraní podle svých potřeb, ale také zvýšila efektivitu práce s aplikací.

4.1.9 SweetAlert2

SweetAlert2[22] je moderní knihovna pro JavaScript, která nabízí bohaté možnosti pro zobrazení upozornění a dialogových oken. Tato knihovna překonává omezení standardních dialogů JavaScriptu tím, že poskytuje široké možnosti pro přizpůsobení vzhledu a interakcí. Dialogová okna vytvořená pomocí SweetAlert2 jsou nejen vizuálně atraktivní, ale také plně responzivní, což zaručuje jejich správnou funkčnost na různých zařízeních a obrazovkách.

Díky SweetAlert2 lze snadno implementovat různé typy upozornění, jako jsou potvrzovací dialogy, výstrahy, chybová hlášení a další, které lze dále rozšířit o prvky jako tlačítka, vstupní pole a animace. Tato flexibilita umožňuje vývojářům vytvářet komplexní uživatelské interakce, které jsou intuitivní a esteticky příjemné.

V projektu je SweetAlert2 využit k poskytování dynamických zpětných vazeb uživatelům. To zahrnuje upozornění na úspěšné dokončení úlohy, varování před

[21] https://www.npmjs.com/package/react-beautiful-dnd
[22] https://sweetalert2.github.io/

potenciálními problémy a žádosti o potvrzení důležitých akcí. Použití SweetAlert2 výrazně zlepšilo vizuální kvalitu těchto interakcí a zároveň zvýšilo celkovou uživatelskou spokojenost díky jasnější komunikaci a lepší vizuální prezentaci.

4.1.10 GitLab

Pro správu verzí a vývoj celé aplikace jsem zvolil platformu GitLab. Hlavním důvodem výběru GitLabu byla jeho dostupnost pod univerzitním účtem. GitLab, na rozdíl od GitHubu, nabízí robustnější sady nástrojů pro správu projektů, což zahrnuje i lepší podporu pro soukromé repozitáře a pokročilé možnosti sledování problémů a automatizace. Dále GitLab umožňuje detailní konfiguraci přístupových práv, což umožnilo snadnou revizi a testování vedoucím práce bez nutnosti nedokončený projekt zveřejnit. Využití GitLabu také přispělo k efektivnímu řízení vývojového cyklu a podpořilo kontinuální integraci a nasazování, což bylo zásadní pro úspěšné dokončení projektu.

4.2 Zjednodušený uzávěr množiny funkčních závislostí

Program provádí výpočet zjednodušeného uzávěru místo celého F+ z několika důvodů. Zjednodušený uzávěr poskytuje dostatečnou informaci pro určení všech potřebných závislostí bez nutnosti odvozovat všechny závislosti, včetně mnoha triviálních. Kompletní F+ by svojí velikostí mohl být pro uživatele spíše matoucí. Vybírat vhodnou závislost pro další krok dekompozice mezi kompletním výčtem všech odvoditelných funkčních závislostí by bylo zdlouhavé. I jeho samotné generování by bylo pro větší počet atributů časově náročné.

4.2.1 Motivace

V průběhu každého kroku dekompozice je formálně potřeba spočítat uzávěr množiny funkčních závislostí F^+ pro danou relaci, následně do vzniklých podrelací přidat všechny funkční závislosti, využívající pouze v nich dostupné atributy, a množiny opět minimalizovat. Celý uzávěr množiny funkčních závislostí je exponenciálně velký vzhledem k počtu atributů relace. Zdaleka ne všechny závislosti jsou ale podstatné pro správnou dekompozici relace, a zdaleka ne všechny závislosti se mohou během dekompozice ztratit.

Pokud budeme uvažovat již uváděnou relaci, kde $A = \{c = \text{ContractId}, s = \text{SupplierId}, j = \text{ProjectId}, d = \text{DeptId}, p = \text{PartId}, q = \text{Quantity}, v = \text{Value}\}$

a F = {$c \rightarrow s,j,d,p,q,v$; $s,d \rightarrow p$; $p \rightarrow d$; $j,p \rightarrow c$; $j \rightarrow s$} z příkladu 2.2.12.2 výše, program Functional Dependencies Checker vygeneruje uzávěr F^+ ve tvaru {c}→{c,d,j,p,q,s,v}, {c}→{c,d,j,p,q,s}. {c}→{c,d,j,p,q,v}, …, {c}→{c,v}, {c}→{c} [Trivial], {c}→{d,j,p,q,s,v}, {c}→{d,j,p,q,s}, {c}→{d,j,p,q,v}, …, {c,d,j,p,q,s,v}→{s,v} [Trivial], {c,d,j,p,q,s,v}→{s} [Trivial], {c,d,j,p,q,s,v}→{v} [Trivial], obsahující celkem 11 541 funkčních závislostí. Triviální závislosti, kde na pravé straně je podmnožina strany levé, nejsou potřeba. Nelze je použít pro dekompozici, a vždy se v uzávěru zachovají. Jejich odstraněním zbyde 9 482 funkčních závislostí. Dále nejsou potřeba závislosti, které mají na pravé straně nějaký – jakýkoliv – atribut z levé strany, například {c}→{c,d,j,p,q,s,v}. Dále nejsou potřeba závislosti, které mají na levé straně ostrou nadmnožinu klíče, například {c,d,j,p,q,s,v}→{s,v}. Ze všech zbylých závislostí se stejnou levou stranou dále stačí jen ta závislost, která má levou stranu maximální možnou. Například ze 63 závislostí {c}→{d,j,p,q,s,v}, {c}→{d,j,p,q,s}, {c}→{d,j,p,q,v}, … {c}→{s,v}, {c}→{s}, {c}→{v} stačí uchovat tu první, tedy závislost {c}→{d,j,p,q,s,v}. Ostatní jsou z ní snadno odvoditelné s použitím tranzitivity s nějakou triviální závislostí {d,j,p,q,s,v}→X, X ⊂ {d,j,p,q,s,v}. Pokud se dále zredukují levé strany závislostí, a závislosti se stejnou levou stranou se spojí do závislosti jediné, zbyde následující množina:

{c}→{s,j,d,p,q,v}	na klíči {*c*} závisí všechny zbývající atributy
{j}→{s}	zadáno v F
{p}→{d}	zadáno v F
{s,d}→{p}	zadáno v F
{j,d}→{s,p,c,q,v}	na klíči {*j*,*d*} závisí všechny zbývající atributy
{j,p}→{d,c,s,q,v}	na klíči {*j*,*p*} závisí všechny zbývající atributy

Formálně lze algoritmus popsat následovně:

4.2.2 Algoritmus

algorithm **SimplyfiedClosure**(
set of dependencies F,
set of attributes A
): **returns set** F^+
ClosureF := ∅;
for each X ⊆ A **do**
Y := **AttributeClosure**(X);
if X ≠ ∅ **and** Y-X ≠ ∅ **then**
ClosureF := ClosureF
∪ {**GetReducedAttributes**(F, X → Y-X) → Y-X}
endif
endfor
merge all FD's in ClosureF by their left sides;
return ClosureF;

5 Uživatelská příručka

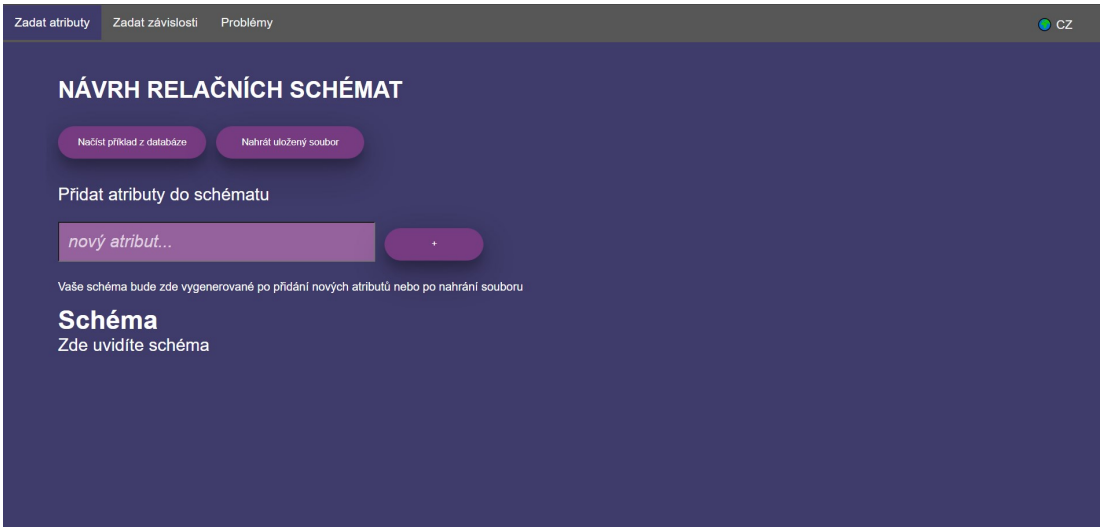
Tato kapitola obsahuje uživatelskou příručku webové aplikace. Aplikace je určená k vizualizaci a interaktivnímu zkoušení algoritmů, důležitých pro návrh relačních databázových schémat. Je navržena tak, aby poskytovala studentům, vyučujícím a dalším uživatelům se zájmem o databázové systémy komplexní nástroj pro studium a pochopení klíčových konceptů, spojených s návrhem a analýzou relačních databází. Běžící aplikace je volně dostupná na adrese http://tirpitz.ms.mff.cuni.cz:3000/.

5.1 Účel příručky

Tato příručka slouží jako ucelený zdroj informací o tom, jak aplikaci efektivně využívat pro dosažení maximálního přínosu.

5.2 Uživatelské desktopové rozhraní

5.2.1 Hlavní stránka



Obrázek 8 - Hlavní stránka webové aplikace

Když navštívíte webovou stránku, kde aplikace běží, zobrazí se Vám hlavní stránka aplikace. Na horní části obrazovky se nachází navigační lišta, která usnadňuje orientaci v programu. Prvním krokem je nahrání nebo zadání atributů. To provedete prostřednictvím příslušné sekce v navigační liště. Následně v sekci "Zadat závislosti" specifikujte závislosti mezi atributy. V sekci "Problémy" si poté můžete vybrat konkrétní problém, který chcete řešit nebo si lépe osvojit. Aplikace podporuje jazykovou lokalizaci, a umožňuje Vám přepínat se mezi českým a anglickým

jazykem. To lze učinit pomocí tlačítka, umístěného v pravém horním rohu obrazovky.

5.2.2 Okna pro návrh relačních schémat

5.2.2.1 Zadávání atributů

Existují tři různé způsoby, jak můžete do systému zadat nebo nahrát schéma relace. Můžete využít databázi příkladů, kterou program nabízí, nahrát vlastní soubor, obsahující dříve uložené schéma relace s množinu funkčních závislostí, nebo zadat potřebné údaje manuálně. Pokud nahrajete správně formátovaný soubor, program zobrazí na obrazovce načtené schéma relace a na další obrazovce zobrazí seznam funkčních závislostí. Následně máte možnost manuálně upravovat jak samotné schéma, tak závislosti. Pro manuální zadání nového atributu využijte pole "Nový Atribut", umístěné v horní části obrazovky, kde můžete zadat atribut o maximální délce deset znaků. Po stisknutí tlačítka "+" se atribut přidá do seznamu. Systém umožňuje zadat maximálně patnáct atributů, což by mělo být dostatečné pro běžné použití.

Pokud po manuálním zadání nahrajete soubor, nebo načtete příklad z databáze, aktuální zadání bude smazáno a nahrazeno daty, které byly načteny ze souboru.

Při pokusu o nahrání soubor ve špatném formátu se zobrazí chybové hlášení.

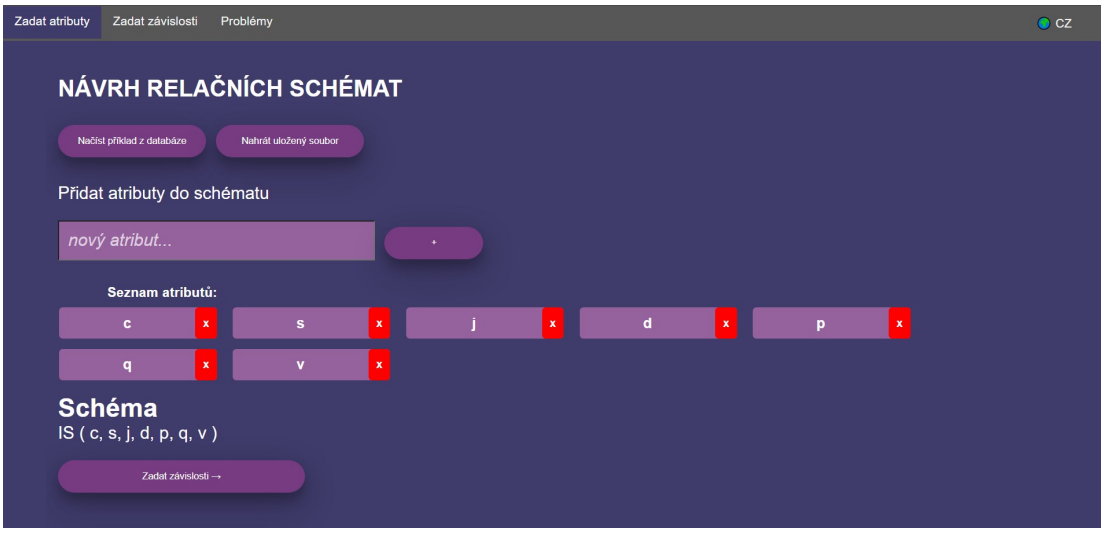
Mějme ukázkový příklad, převzatý ze zveřejněného zkouškového testu:

NÁVRH RELAČNÍCH SCHÉMAT

Schéma: IS (*c, s, j, d, p, q, v)*

Závislosti: F = {$c \rightarrow c,s,j,d,p,q,v$; $s,d \rightarrow p$; $p \rightarrow d$; $j,p \rightarrow c$; $j \rightarrow s$}

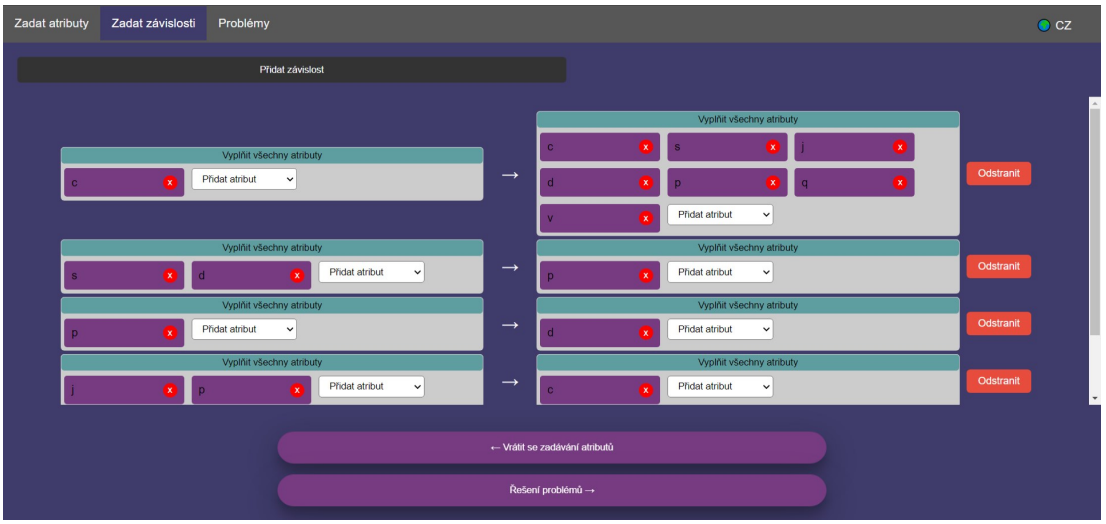
Po zadání schématu uvidíte seznam atributů tak, jak je znázorněno na následujícím obrázku.



Obrázek 10 - Obrazovka pro zadávání atributů do schématu

Po přidání každého atributu do schématu pomocí tlačítka "+" u textového pole se atribut zobrazí jako značka (tag) pod ním. Pomocí tlačítka "×" je možné atribut ze schématu odstranit. Schéma se automaticky měnit s každým přidáním nebo odebráním atributu. Po zadání prvního atributu se zobrazí tlačítko "Zadat závislost". Toto tlačítko opět zmizí, pokud ze schématu odstraníte poslední atribut. Pořadí atributů je možné libovolně upravovat pomocí myši. Můžete tak atribut snadno uchopit a přemístit jej na požadované místo v seznamu.

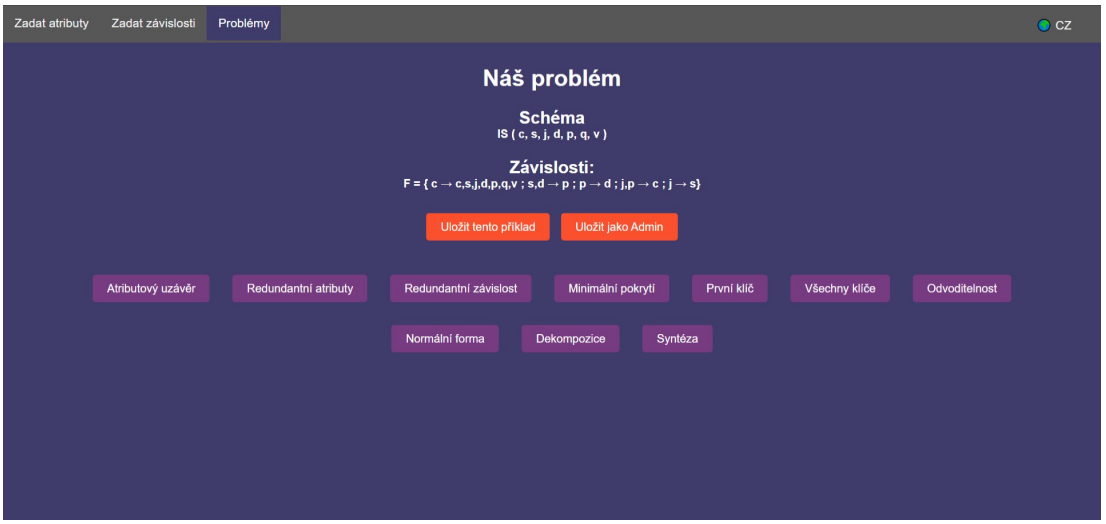
5.2.2.2 **Zadávání závislostí**



Obrázek 11 - Obrazovka pro zadávání závislostí

Po kliknutí na tlačítko „Zadat závislosti“ budete přesměrován do sekce „Zadat závislosti“, kde může specifikovat funkční závislosti mezi jednotlivými atributy a jejich skupinami. Atribut přidáte pomocí tlačítka “+”. Následně si ve zobrazeném select boxu vyberete atribut, který potřebujete. Select box se dá odstranit, stejně jako se dá změnit jeho hodnota. Pro přidání dalších funkčních závislostí slouží tlačítko “Přidat závislost”. Pořadí závislostí je opět možné libovolně upravit pomocí techniky "drag and drop". Funkční závislost Tak můžete snadno uchopit myší a přetáhnout ji na požadované místo v seznamu.

5.2.3 Okno pro volbu problému k vyřešení



Obrázek 12 - Obrazovka pro výběr problému k vyřešení

V sekci "Problémy" se nachází schéma a zadané funkční závislosti. Tato sekce rovněž umožňuje ukládat definici schématu spolu s funkčními závislostmi problémy na Váš počítač ve formátu .txt souboru pomocí tlačítka "Uložit soubor". Vedle tohoto tlačítka se nachází tlačítko "Uložit na server", které je určeno výhradně pro správce stránky. Správce může pomocí tohoto tlačítka ukládat definice přímo do databáze na serveru, čímž definici zpřístupní všem ostatním uživatelům. Pro ukládání na server je vyžadováno zadání hesla.

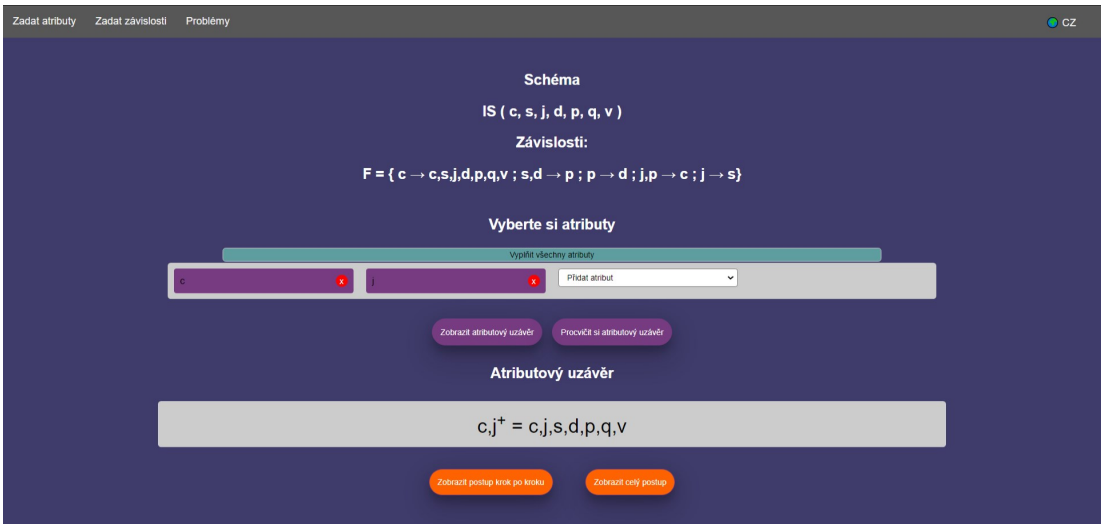
Po zadání atributů a závislostí máte možnost si zobrazit řešení, vyzkoušet si vyřešit, případně si nechat vysvětlit řešení různých problémů na d zadaným schématem. K dispozici jsou algoritmy:

1. “***Atributový uzávěr***” (2.2.3):
   Nalezení a zobrazení všech atributů, které jsou funkčně závislé na zadané množině atributů.

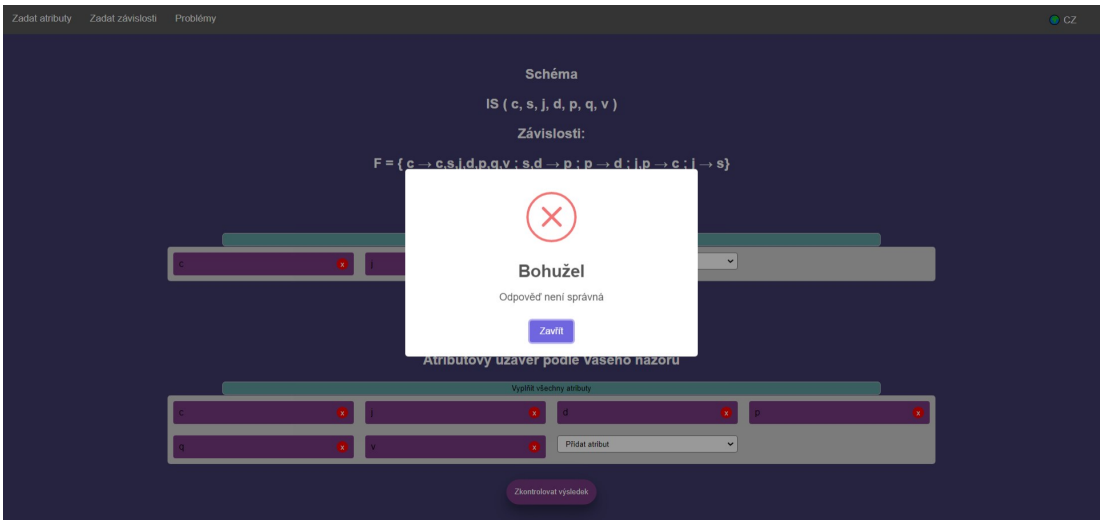
2. “***Redundantní atributy***” (2.2.6):
   Nalezení a odstranění všech redundantních atributů na levé straně funkčních závislostí a vrácení množiny funkčních závislostí bez redundantních atributů.
3. “***Redundantní závislosti***” (2.2.8):
   Nalezení a odstranění všech redundantních závislostí v množině funkčních závislostí a vrácení neredundantní množiny funkčních závislostí.
4. “***Minimální pokrytí***” (2.2.9):
   Nalezení a zobrazení minimálního pokrytí původní množiny funkčních závislostí.
5. “***První klíč***” (2.2.10):
   Nalezení a zobrazení prvního klíče zadaného schématu.
6. “***Všechny klíče***” (2.2.10):
   Nalezení a zobrazení všech klíčů zadaného schématu. Zobrazování všech klíčů bude možné i pokud předtím nebyl vygenerován první klíč.
7. “***Normální forma***” (2.2.11):
   Určení, ve které normální formě se nachází zadané schéma.
8. “***Odvoditelnost***” (2.2.4):
   Poskytuje uživatelům možnost ověřit, zda lze specifické funkční závislosti odvodit z dalších závislostí, které jsou již definovány v schématu.
9. “***Dekompozice***” (2.2.12):
   Zobrazení rozdělení univerzální relace algoritmem dekompozice do BCNF.
10. “***Syntéza***” (2.2.13):
    Zobrazení rozdělení relace algoritmem syntéza do 3NF.

5.2.3.1 Atributový uzávěr

V sekci pro nalezení atributového uzávěru je zobrazeno zadané relační schéma. Máte možnost zadat libovolnou podmnožinu existujících atributů. Po kliknutí na tlačítko "Zobrazit atributový uzávěr" se zobrazí nejen výsledek, ale i dvě další tlačítka, která umožňují buď zobrazit detailní postup výpočtu, nebo postup, potřebný pro dosažení daného výsledku krok za krokem.



Obrázek 13 - Okno pro řešení problému atributový uzávěr



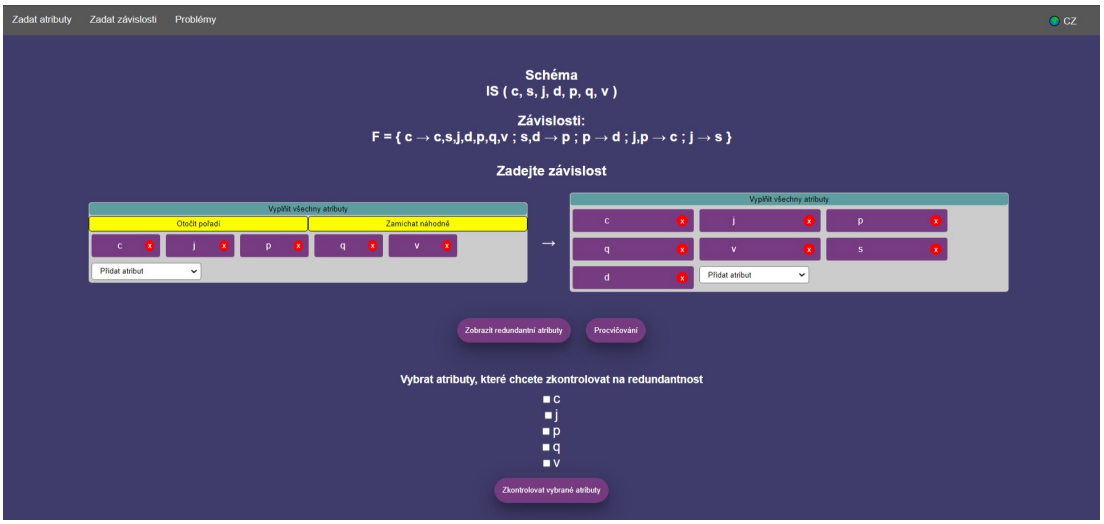
Obrázek 14 - Procvičování atributového uzávěru

Naleznete zde rovněž tlačítko „Procvičit si atributový uzávěr“, které Vám umožňuje aktivně si ověřit své znalosti. Po kliknutí na toto tlačítko je umožněno interaktivně vložit vlastní řešení a následně si zkontrolovat správnost odpovědi.

5.2.3.2 Redundantní atributy

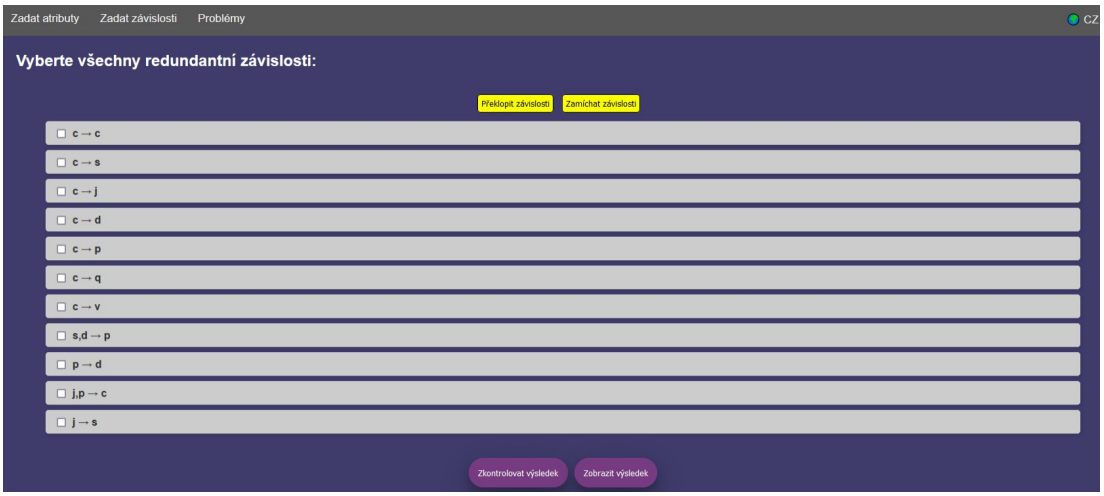
Aplikace s využitím atributových uzávěrů poskytuje rovněž možnost práce s redundantními atributy, dostupné pomocí tlačítka „Zobrazit redundantní atributy“. Umožňuje identifikovat redundantní atributy na levé straně zadané funkční závislosti. Manipulace s pořadím atributů na levé straně funkční závislosti přímo ovlivňuje výsledek, což umožňuje hlouběji pochopit, jak redundance atributů na levé straně závisí na pořadí, ve kterém jsou atributy na redundanci testovány. Pokud si chcete procvičení potřebného postupu, můžete využít tlačítko „Procvičování“. Zde si

můžete vybrat atributy, které považujete za redundantní, a pomocí tlačítka „Zkontrolovat vybrané atributy“ ověřit správnost svého řešení.



Obrázek 15 - Okno pro řešení problému redundantních atributů

5.2.3.3 Redundantní závislosti



Obrázek 16 - Okno pro řešení problému redundantních závislostí

V sekci redundantní závislosti můžete ze zadaných závislostí vybrat ty, které považujete za redundantní. Program poskytuje funkci automatického ověření, při kterém probíhá identifikace redundantních závislostí od shora dolů. To zajišťuje sekvenční kontrolu podle zadaného pořadí. Máte možnost sami označit závislosti, které považujete za redundantní, a následně nechat program ověřit jejich výběr. Klíčovým prvkem pro zajištění přesnosti analýzy je možnost přeuspořádat závislostí, díky čemuž lze ovlivnit výsledek a ozřejmit si, jak spolu pořadí kontroly závislostí a výsledek souvisí.

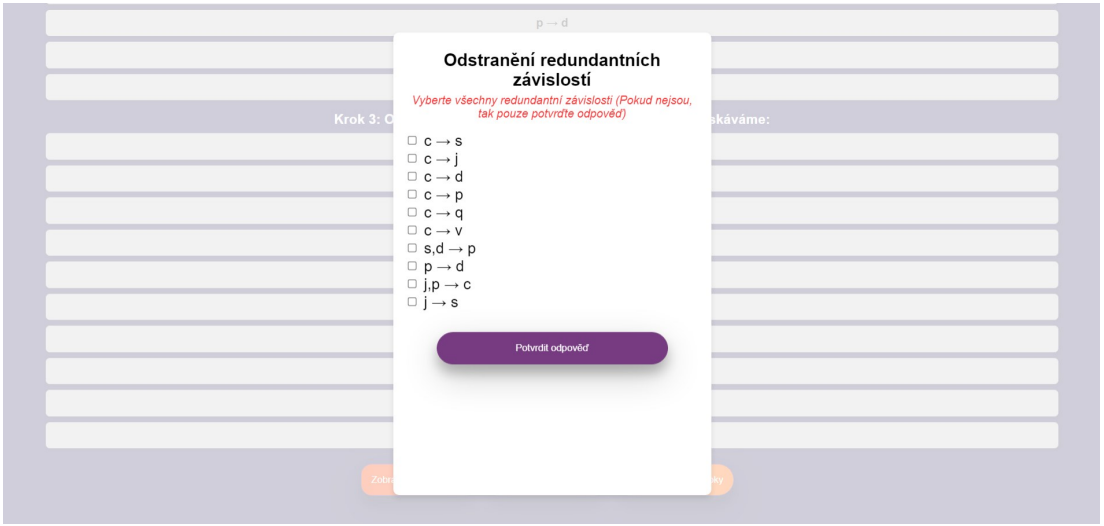
5.2.3.4 Minimální pokrytí

Obrázek 17 - Okno pro řešení problému minimálního pokrytí

V rámci vizualizace procesu získávání minimálního pokrytí množiny funkčních závislostí je Vám prezentován jeden z možných výsledků. Vzhledem k tomu, že finální výsledek může záviset na pořadí, ve kterém jsou funkční závislosti zpracovávány, aplikace Vám opět umožňuje interakci s pořadím jejich zpracování. Pro hlubší porozumění procesu jsou dostupná dvě tlačítka: „Zobrazit krok po kroku“ a „Zobrazit všechny kroky“.

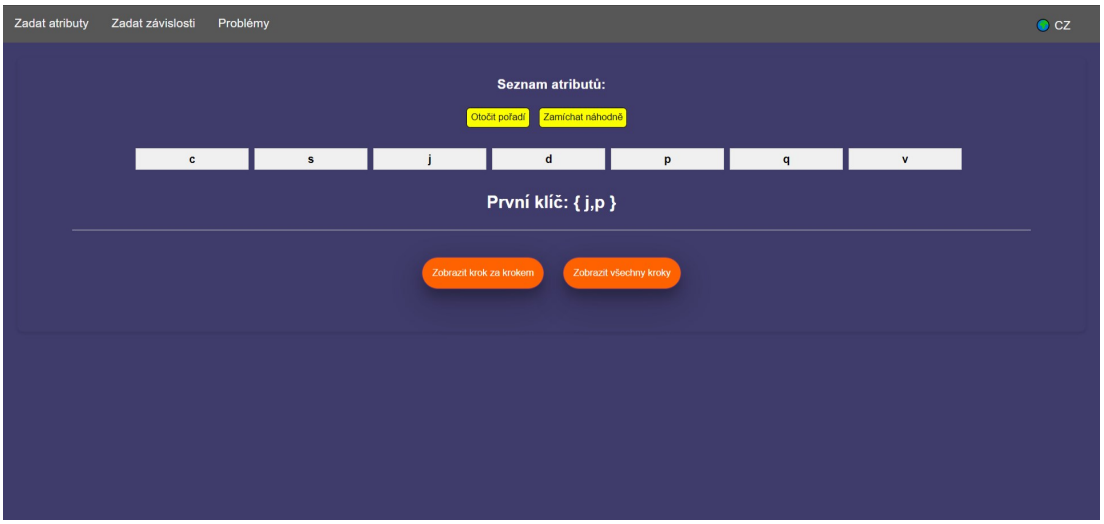
Po kliknutí na „Zobrazit krok po kroku“ se zobrazí první krok procesu, kde jsou všechny závislosti rozepsány na elementární, tedy tak, aby na pravé straně každé závislosti byl pouze jeden atribut. Máte možnost modifikovat pořadí závislostí a tak ovlivnit průběh dalších kroků. Tento interaktivní prvek umožňuje experimentovat s různými scénáři a vidět, jak se změny v pořadí závislostí promítnou do výsledného minimálního pokrytí.

V dalších krocích Vám aplikace nabízí možnost samostatně určit, které kroky budou následovat, s cílem ozřejmit Vám, jak se z jednotlivých kroků skládá celkový proces minimalizace. Kliknutím na „Zobrazit všechny kroky“ se zobrazí kompletní proces dosažení minimálního pokrytí bez nutnosti interakce, což je vhodné pro rychlý přehled nebo revizi výsledku.



Obrázek 18 - Interaktivní rozhraní pro samostatné procvičování minimalizace funkčních závislostí

5.2.3.5 První klíč



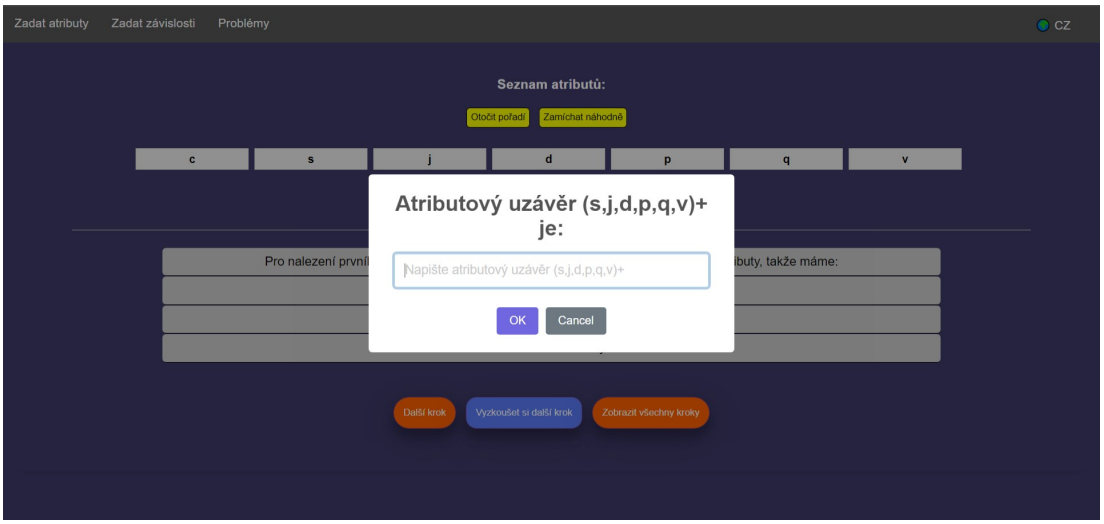
Obrázek 19 - Okno pro řešení problému jak najít první klíč

V sekci "První klíč" je ihned po otevření zobrazen jeden z klíčů. Opět platí, že nalezený klíč může záviset na pořadí, ve kterém jsou atributy odstraňovány. Pro zvýšení interaktivity a pochopení tohoto procesu jsou v horní části obrazovky uvedeny všechny relevantní atributy, které uživatel můžete přesouvat a tak experimentálně měnit jejich pořadí. Jakmile přesunete jakýkoli atribut, výsledek se okamžitě aktualizuje. Zobrazený klíč se tak může změnit.

Pro detailnější analýzu procesu hledání prvního klíče jsou k dispozici dvě tlačítka: "Zobrazit krok za krokem" a "Zobrazit všechny kroky". Výběrem "Zobrazit všechny kroky" obdržíte kompletní seznam všech kroků, které algoritmus podnikl k dosažení

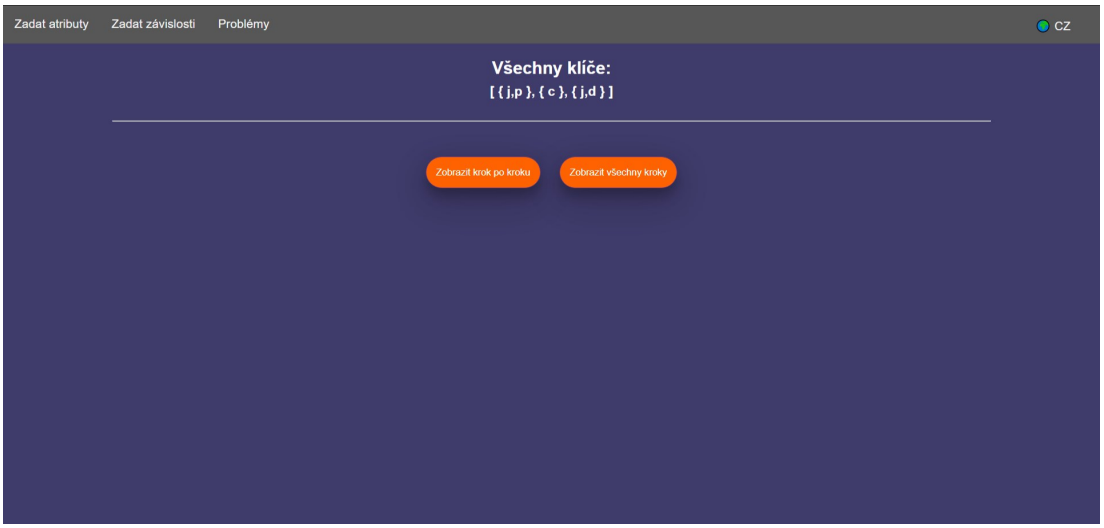
výsledku. Toto je užitečné pro rychlý přehled nebo pokud potřebujete porozumět celkové metodice procesu.

Kliknutím na tlačítko "Zobrazit krok za krokem" se kroky zobrazují postupně a máte tak možnost interaktivně zasahovat do procesu. V určitém kroku můžete rovněž zadat svůj odhad dalšího kroku a systém následně vyhodnotí, zda je jeho odpověď správná. Tato metoda Vám poskytne praktické zkušenosti a pomáhá lépe porozumět logice a strategii odstraňování atributů při hledání klíče.



Obrázek 20 - Interaktivní rozhraní pro samostatné procvičování hledání prvního klíče

5.2.3.6 Všechny klíče



Obrázek 21 - Okno pro řešení problému jak najít všechny klíče

V sekci "Všechny klíče" jsou po otevření zobrazeny všechny klíče zadané relace. Aby bylo možné detailněji prozkoumat postup, jakým byly tyto klíče získány, jsou k dispozici dvě tlačítka: "Zobrazit krok za krokem" a "Zobrazit všechny kroky".

Kliknutím na "Zobrazit všechny kroky" získáte úplný přehled o všech krocích procesu, které vedly k odvození všech klíčů. Toto je ideální pro rychlé získání přehledu o celkovém procesu bez nutnosti interaktivní účasti.

Naopak, volba "Zobrazit krok za krokem" umožňuje postupovat krok po kroku skrze algoritmus hledání klíčů. V tomto režimu můžete také zadávat vlastní návrhy na další krok a systém následně validuje, zda byl Váš odhad správný. Tento interaktivní přístup nejenže podporuje hlubší porozumění problematice, ale také umožňuje uživatelům aktivně se zapojit do procesu učení a testování vlastních znalostí.

5.2.3.7 Odvoditelnost

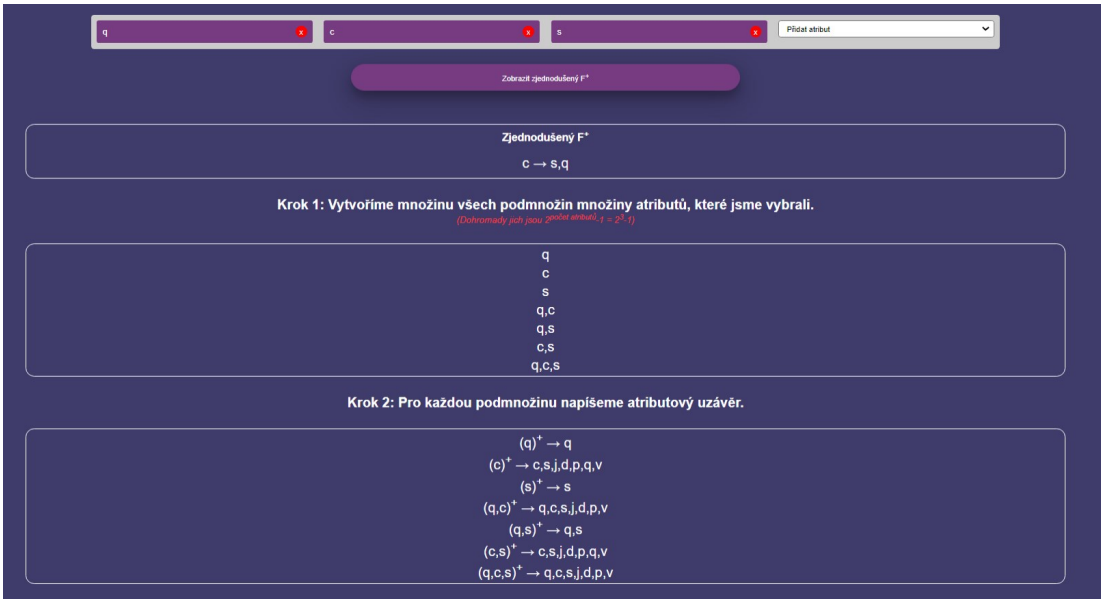
Obrázek 22 - Okno pro řešení problému odvoditelnost

V sekci "Odvoditelnost" jsou zobrazené již zadané atributy a závislosti. Uživatelé zde mají možnost ověřit odvoditelnost specifických funkčních závislostí pomocí tlačítka "Zkontrolovat odvoditelnost". Po aktivaci tohoto tlačítka systém okamžitě reaguje odpovědí "ano" nebo "ne", čímž uživatelé získají zpětnou vazbu na správnost svých úvah. Tato funkce umožňuje uživatelům rychle a efektivně ověřit, zda jejich pochopení vztahů mezi atributy odpovídá logice databázového schématu.

Důležitým prvkem této sekce je také možnost generování *zjednodušeného uzávěru množiny funkčních závislostí* F^+. viz sekce 4.2 Zjednodušený uzávěr množiny funkčních závislostí.

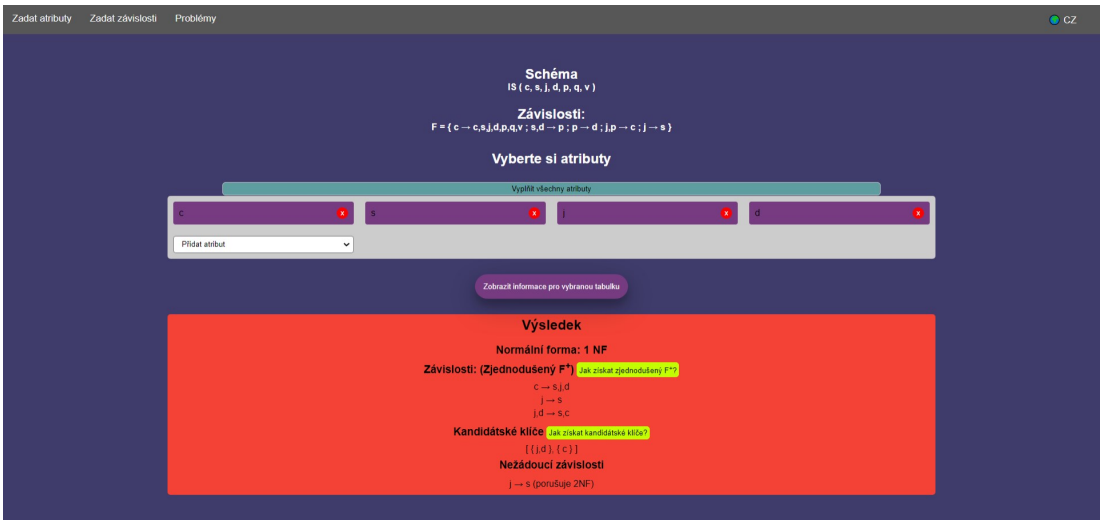
Je možné vybrat atributy, pro které chcete zjednodušený uzávěr získat. Výsledek obsahuje závislosti odvoditelné pro množinu vybraných atributů, vhodných, případně nezbytných pro dekompozici a určení normálních forem. Po vygenerování

zjednodušeného uzávěru máte na výběr zobrazení celého procesu buďto najednou, nebo krok za krokem. V režimu "zobrazit krok po kroku" však není možné procvičování. Jde o lineární demonstraci postupu.



Obrázek 22 - Okno pro zobrazující postup jak získat zjednodušené F^+

5.2.3.8 Normální forma



Obrázek 23 - Okno pro řešení problému normální forma

V sekci "Normální forma" je zobrazeno zadané schéma a závislosti. Z množiny atributů je možné si vybrat specifické atributy, a zjistit, jakou normální formu zvolená podrelace spolu s aplikovatelnými závislostmi splňuje. Kromě normální formy je k dispozici zjednodušený uzávěr množiny závislostí., seznam kandidátních

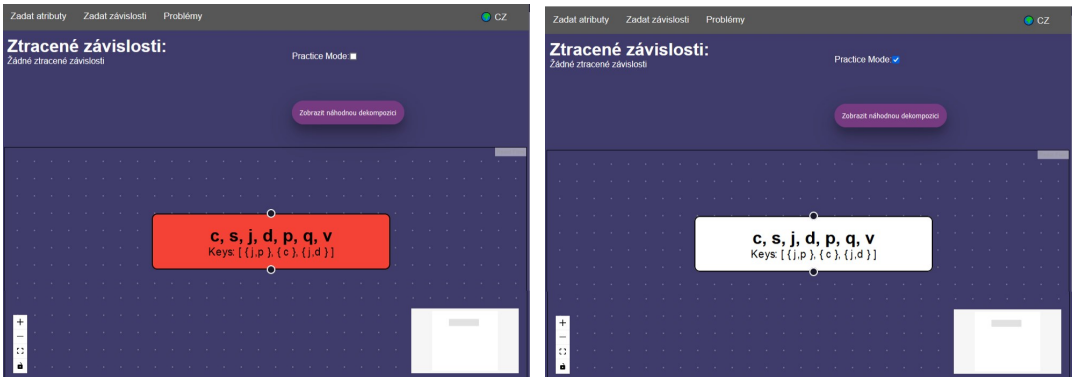
klíčů a identifikace problematických závislosti, které brání dosažení Boyce-Coddovy normální formy (BCNF).

Pro snadnou orientaci v tom, jakou nejvyšší normální formu relace splňuje, je výsledek podbarven. Barvy jsou inspirovány semaforem. 1NF je zvýrazněna červenou, 2NF žluto-oranžovou, 3NF tmavě zelenou a BCNF světle zelenou barvou. Toto barevné kódování nejen zlepšuje srozumitelnost výsledků, ale také podporuje rychlé a intuitivní pochopení splněné normální formy pro libovolnou vybranou podmnožinu atributů (podrelaci).

5.2.3.9 Dekompozice

V sekci "Dekompozice" je zobrazena univerzální relace, znázorněná jediným uzlem, obsahujícím všechny zadané atributy se všemi zadanými funkčními závislostmi. Zobrazeny jsou také všechny kandidátní klíče této relace.

Sekce nabízí dva navzájem se doplňující režimy práce s relacemi. Ve výchozím režimu aplikace umožňuje relace v nevyhovující normální formě postupně rozdělovat na menší. V procvičovacím režimu si můžete ověřovat své porozumnění normálním formám u již vytvořených relací, ve druhém režimu V tomto režimu je uzel (relace) opět podbarven dle nejvyšší splněné normální formy.

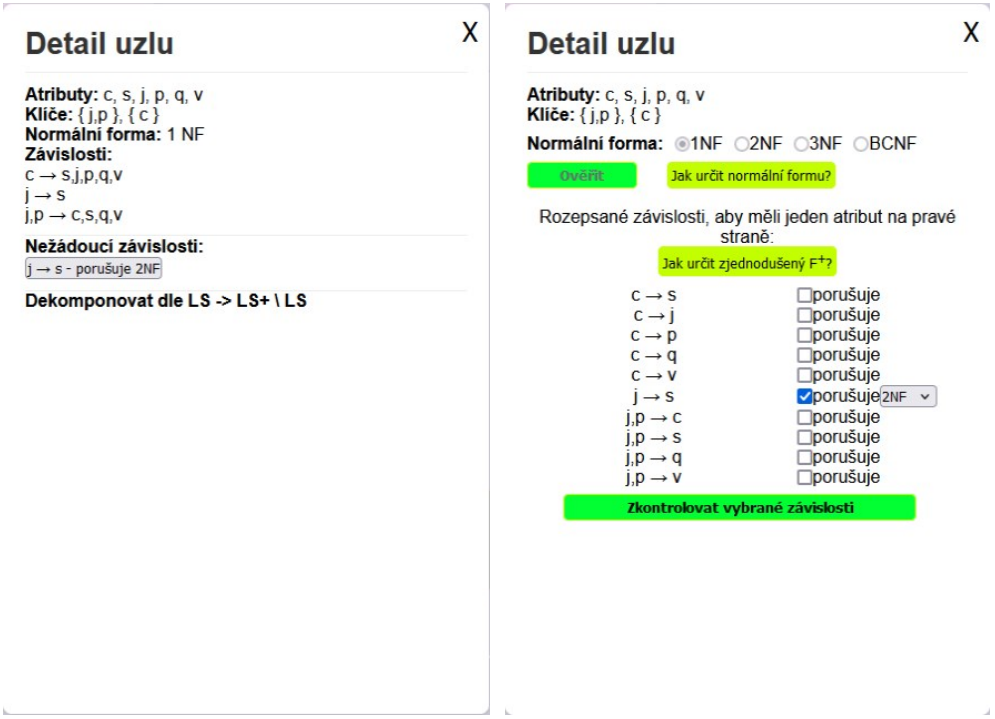


Obrázek 24 - Okno pro řešení problému dekompozice v závislosti na režimu

V horní části okna se nachází tlačítko "Zobrazit náhodnou dekompozici", které umožňuje zobrazit náhodně generovanou dekompozici uzlu. Pro hlubší analýzu můžete dvakrát kliknout na jakýkoliv zobrazený uzel, čímž se otevře dialog s podrobnými informacemi o dané relaci. Ve výchozím režimu informace odpovídá té, která se zobrazuje v sekci “Normální forma”, a zahrnuje seznam obsažených atributů, seznam kandidátních klíčů, aktuální normální formu a zjednodušený uzávěr množiny funkčních závislostí. Zvýrazněny jsou nežádoucí závislosti, které brání

dosažení BCNF. Jakákoliv nežádoucí funkční závislosti může být přímo použita pro další dekompozici. Kliknutím na takovou funkční závislost se podle ní daná relace rozdělí na dvě podrelace. V procvičovacím režimu můžete označit Vámi určenou dosaženou normální formu relace, případně pro jednotlivé závislosti určit, jakou (nejnižší) normální formu porušují.

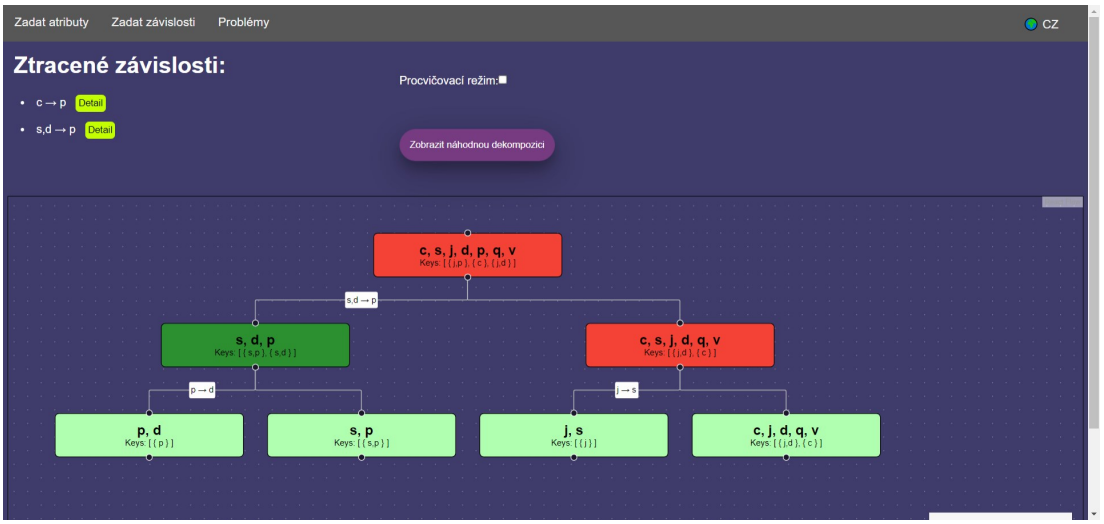
Kromě těchto funkcí systém nabízí také alternativní možnosti rozdělení uzlu, které mohou být vhodnější pro následné kroky dekompozice. Jedná se o závislosti, které pro danou levou stranu uvažují na pravé straně všechny (netriviálně) funkčně závislé atributy, tedy závislosti ve tvaru $LS \rightarrow LS^{+} \setminus LS$, kde LS je jedna z levých stran zadaných závislostí.



Obrázek 25 - Okno s detailními informacemi o relaci v závislosti na režimu

To může pomoci v tom nerozdělit řetěz tranzitivních závislostí uprostřed, a neztratit tak zbytečně nějakou závislost. Navíc, tím, že jsou do jedné relace umístěny všechny atributy, závislé na daném klíči, se zamezí vzniku více relací se stejným klíčem. Celý proces je podpořen interaktivním grafickým rozhraním[23], což umožňuje plynulé ovládání grafu, včetně možnosti přiblížení, oddálení a navigace po grafu.

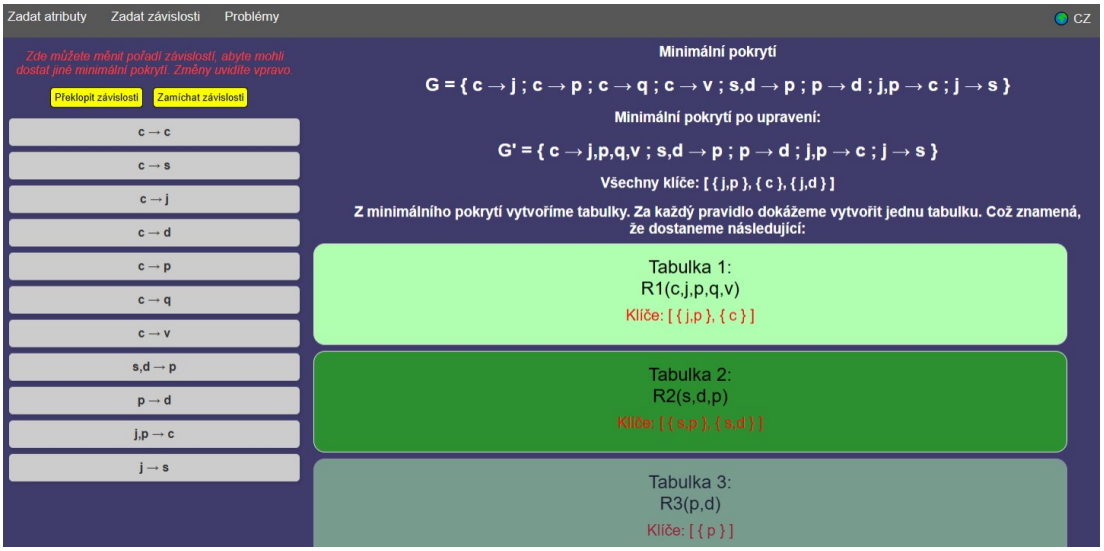
[23] Vyvinutým s využitím knihovny React Flow



Obrázek 26 - Ukázka jedné z možných dekompozic

Během každého rozdělení uzlu či po náhodném vygenerování dekompozice aplikace kontinuálně sleduje, zda nedošlo ke ztrátě nějaké závislostí. Ztracené závislosti, které ale mohou být důležité pro udržení integrity dat a jejich původní sémantiky, jsou okamžitě zobrazeny v levém horním rohu uživatelského rozhraní programu. Toto umístění zajišťuje, že nebudou nepřehlédnuty. Díky tomu si udržíte přehled o tom, nakolik ta která dekompozice zlepšuje normální formu databázového schématu na straně jedné, a zároveň udržuje úplnost a logickou konzistenci funkčních závislostí na straně druhé.

5.2.3.10 Syntéza



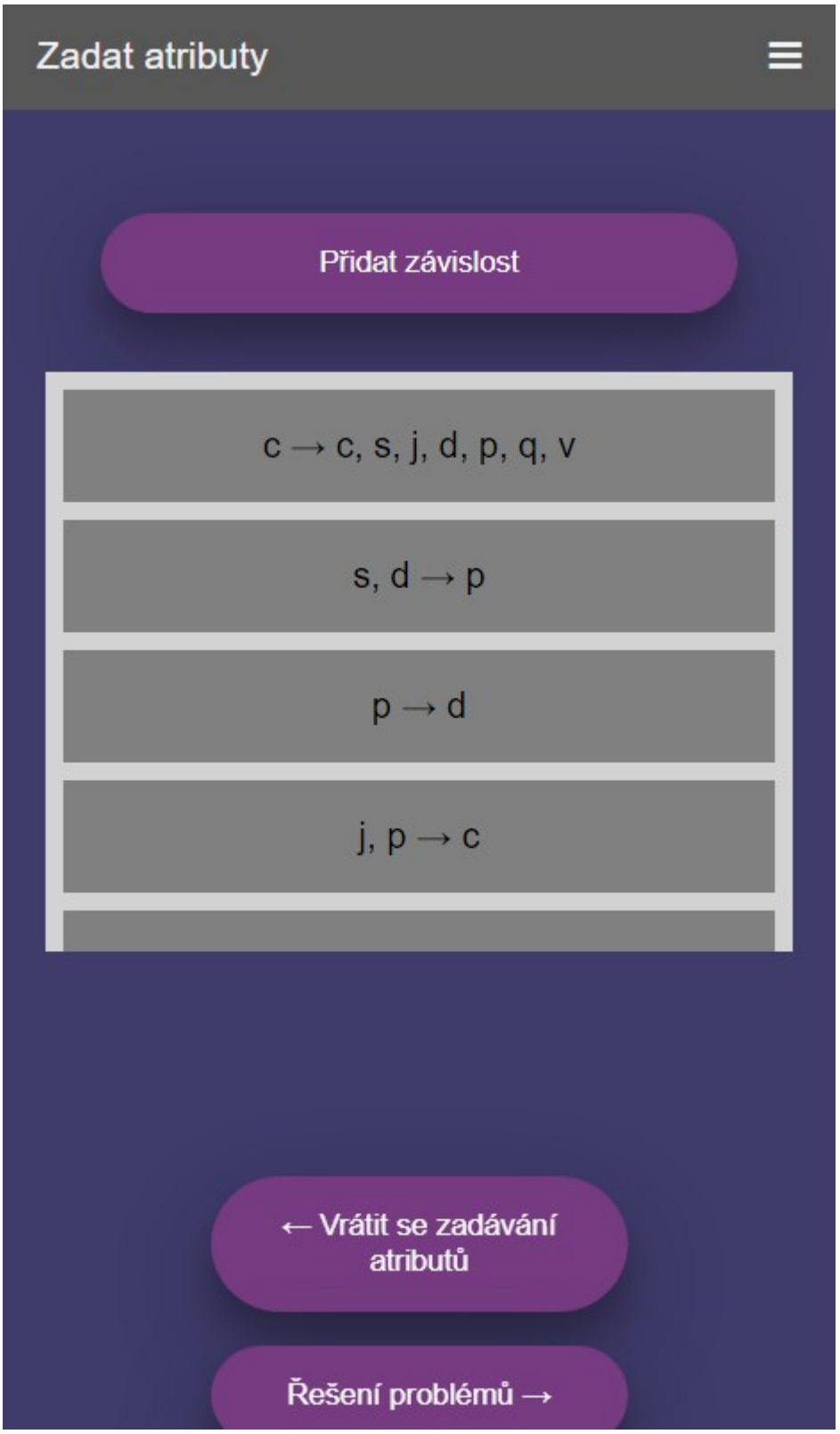
Obrázek 27 - Okno pro řešení problému syntéza

V sekci "Syntéza" je obrazovka rozdělena na levou a pravou část. Na pravé straně se zobrazuje minimální pokrytí množiny funkčních závislostí spolu se všemi kandidátními klíči a dále seznamem uzlů, které reprezentují relace, které byly během syntézy vytvořeny. Každá relace přímo zobrazuje v ní obsažené atributy a kandidátní klíče. Jednotlivé uzly jsou opět jsou podbarveny podle toho, do které normální formy patří. Kliknutím na konkrétní uzel se zobrazí detailnější informace o tabulce, včetně zjednodušeného uzávěru funkčních závislostí. Aplikace také upozorňuje na vzniklé relace, které jsou nadbytečné, protože jsou podrelacemi jiných relací. Rovněž nabízí možnost sloučení tabulek, pokud mají shodné nebo funkčně ekvivalentní klíče. Stačí jednu z relací pomocí myši přetáhnout nad relaci jinou. Aplikace vyhodnotí, zda je sloučení relací možné a pokud není, zobrazí se chybové hlášení.

Na levé straně okna jsou zobrazeny uživatelem zadané funkční závislosti rozepsané na elementární, kde na pravé straně je pouze jeden atribut. Uživatel může tyto závislosti přesouvat a měnit jejich pořadí, čímž ovlivňuje výsledné minimální pokrytí a tedy i výsledek syntézy.

5.3 Uživatelské mobilní rozhraní

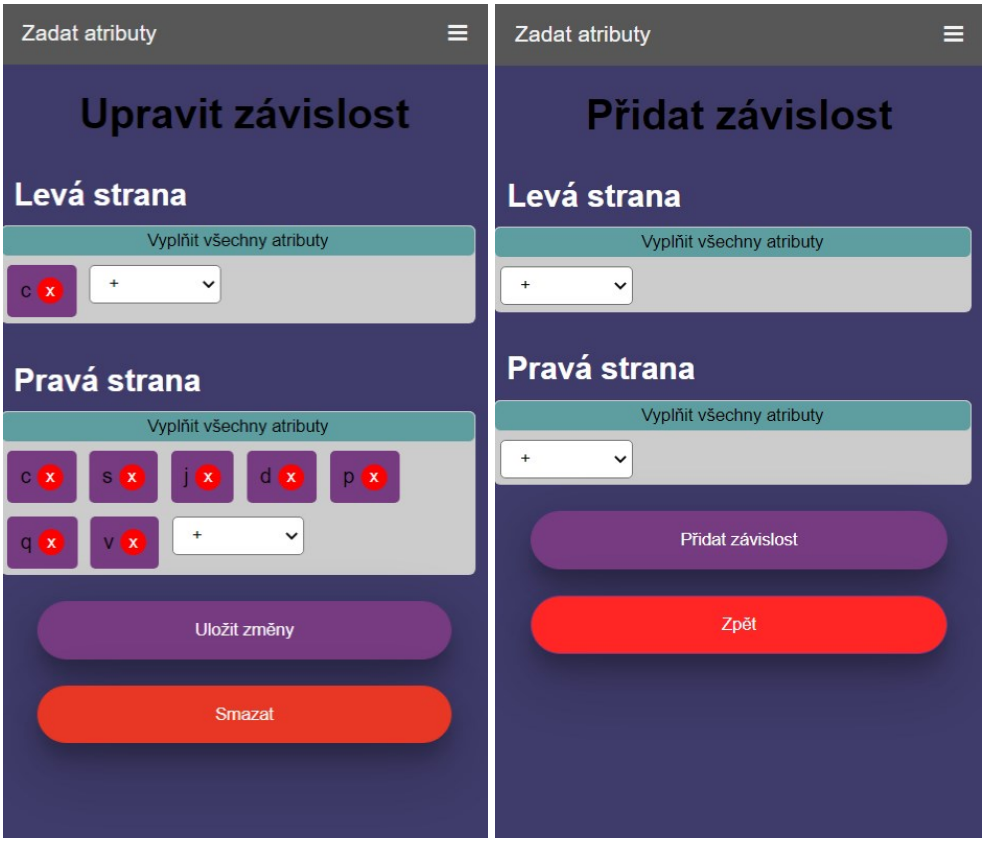
Jako každé moderní aplikace, i tato implementuje mobilní uživatelské rozhraní, založené na principech responzivního designu. I v mobilní verzi jsou zachovány všechny funkce, dostupné ve verzi desktopové. Existuje však jedna specifická odlišnost, která se týká způsobu zadávání funkčních závislostí.



Obrázek 28 - Rozhraní pro zadávání závislostí na mobilním zařízení

V rámci mobilního uživatelského rozhraní můžete přidávat nové závislosti prostřednictvím tlačítka "Přidat závislost", které Vás přesměruje na obrazovku určenou k zadávání nových závislostí. Pro úpravu stávajících závislostí je nutné na

danou závislost dvojitě kliknout, což Vás přesměruje na stránku, kde je možné závislost upravit nebo smazat. Manipulace se závislostmi v mobilním režimu zůstává intuitivní; závislost lze přemístit dlouhým stiskem a následným táhnutím na požadované místo. Tato funkcionalita zajišťuje, že i v mobilním režimu je aplikace plně operativní a umožňuje efektivní správu funkčních závislostí.



Obrázek 29 - Rozhraní mobilního režimu pro přidávání a úpravu závislostí

6 Návrh architektury

Projekt je strukturován v souladu s běžnými konvencemi pro React aplikace, což zahrnuje jasně organizovanou strukturu složek a standardní konfigurační soubory.

Kořenový adresář: Zde se nacházejí konfigurační soubory jako **.gitignore** a **package.json**. Soubor **.gitignore** specifikuje soubory a adresáře, které mají být ignorovány systémem Git, zatímco **package.json** popisuje projekt a jeho závislosti.

Adresář **public/**: Obsahuje statické soubory jako **index.html**, který je hlavním HTML souborem, načítaným při spuštění aplikace.

Adresář **src/**: Tvoří jádro projektu, kde je uložen veškerý zdrojový kód.

- **App.js** je hlavní komponenta React a slouží jako vstupní bod do aplikace.
- Adresář **algorithm/** zahrnuje logiku algoritmů, použitých v aplikaci.
- Adresář **components/** obsahuje React komponenty, dále organizované do podadresářů podle jejich funkce nebo kontextu. Je rozdělen na dvě hlavní části: **navbar/** a **content/**
    - **navbar/:** obsahuje komponenty pro navigační lištu aplikace.
        - **Navbar.jsx**: Hlavní komponenta navigační lištu.
        - **navbar.scss**: Styly pro navigační lištu.
    - **content/:** obsahuje komponenty pro hlavní obsah aplikace.
        - Podadresáře:
            - **attribute/**: Obsahuje komponenty a styly pro práci s atributy.
            - **dependency/**: Obsahuje komponenty a styly pro práci s závislostmi. Dále je rozdělen na mobile/ a pc/ pro různé zobrazení na různých zařízeních.
            - **problems**/: Obsahuje komponenty a styly pro zobrazení problémů.
- Adresář **config/:** zahrnuje konfigurační soubor **firebase.js** pro inicializaci Firebase.
- Adresář **contexts/:** poskytuje React Contexty, které umožňují sdílení globálního stavu napříč aplikací.
- **i18n.js** je konfigurační soubor pro internacionalizaci aplikace s využitím knihovny i18next.
- **index.js** slouží jako vstupní bod JavaScriptu pro aplikaci.

- **constantValues/constantValues.js**: obsahuje konstanty používané v programu.
- Adresář **locales/:** obsahuje překlady pro různé jazyky použité v aplikaci.
- Adresář **styles/:** zahrnuje globální styly aplikace napsané v SCSS.

7 Programátorská příručka

7.1 Minimální požadavky a instalace programu

Předinstalovaná aplikace je dostupná na webové adrese http://tirpitz.ms.mff.cuni.cz:3000/.

Projekt je vytvořen v prostředí React.js a pro jeho spuštění je zapotřebí splnit následující minimální požadavky:

- Node.js: Verze 12.0.0 nebo novější.
- NPM (Node Package Manager): Tento správce balíčků je obvykle instalován spolu s Node.js.

Instalace:

- Stáhněte si zdrojový kód projektu z repozitáře[24]
- Otevřete terminál a přejděte do složky s projektem **my-app/**.
- Spusťte příkaz **npm install** pro instalaci všech závislostí projektu.
- Po úspěšné instalaci závislostí spusťte příkaz **npm start** pro spuštění projektu.

7.2 Databáze

Ve projektu je pro správu a ukládání dat využita cloudová platforma Firebase od společnosti Google. Firebase poskytuje komplexní sadu nástrojů pro efektivní vývoj webových a mobilních aplikací, což zahrnuje i databázové služby.

Konkrétně je použita Firebase Realtime Database, což je cloudová NoSQL databáze, která umožňuje ukládat data ve formátu JSON a synchronizovat je v reálném čase mezi všemi klienty. Tato funkce je zvláště užitečná pro aplikace, vyžadující rychlé aktualizace dat bez nutnosti čekání na odpověď serveru.

Inicializace Firebase v projektu probíhá skrze konfigurační soubor firebase.js, kde je definována konfigurace spojení s databází. Zde je příklad konfigurace:

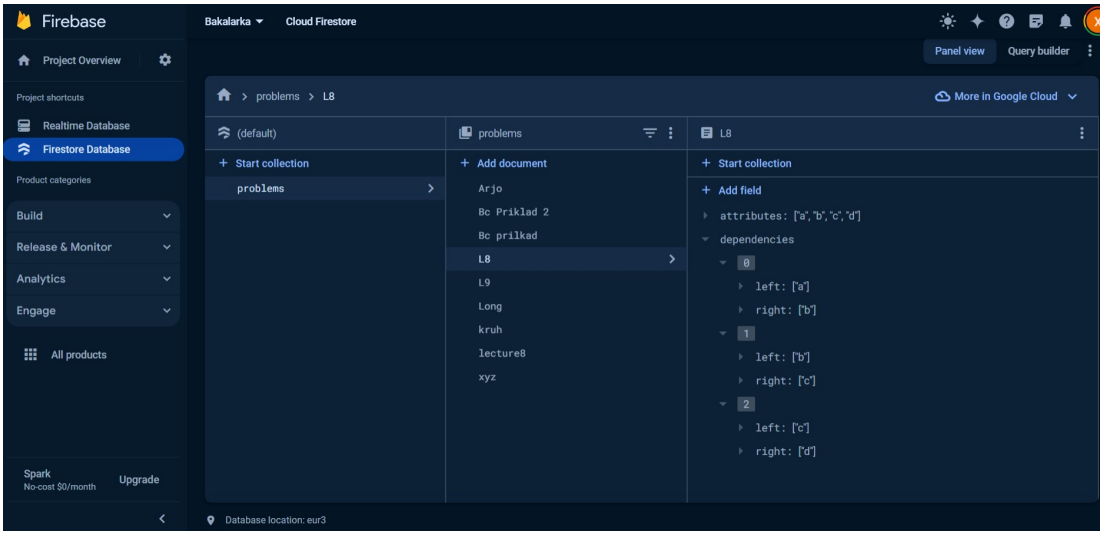
[24] https://gitlab.mff.cuni.cz/teaching/nprg045/kopecky/Duong_Xuan_Anh_2022/

```
import { initializeApp } from "firebase/app";
import { getDatabase } from "firebase/database";

const firebaseConfig = {
  apiKey: "YOUR_API_KEY",
  authDomain: "YOUR_PROJECT_ID.firebaseapp.com",
  databaseURL: "https://YOUR_PROJECT_ID.firebaseio.com",
  projectId: "YOUR_PROJECT_ID",
  storageBucket: "YOUR_PROJECT_ID.appspot.com",
  messagingSenderId: "YOUR_SENDER_ID",
  appId: "YOUR_APP_ID",
  measurementId: "YOUR_MEASUREMENT_ID"
};

const app = initializeApp(firebaseConfig);
export const db = getDatabase(app);
```

Obrázek 30 - struktura konfiguračního souboru firebase.js

Firebase Realtime Database nabízí vysokou míru flexibility, což umožnilo přizpůsobit databázi specifickým požadavkům projektu. Konfigurační soubor firebase.js, který obsahuje údaje pro připojení k databázi, je navržen tak, aby byl snadno upravitelný. Programátor může jednoduše změnit údaje v konfiguraci a použít tak vlastní instanci Firebase Realtime Database. Toto může být zvláště užitečné v případech, kdy jsou požadavky na zabezpečení, kapacitu úložiště nebo regionální dostupnost odlišné od výchozích nastavení.

7.2.1 Struktura Firestore Databáze



Obrázek 31 - Ukázka FireStore databáze

V rámci projektu je pro ukládání definic problémů, vztahujících se k návrhu relačních schémat v relačních databázích ve Firestore databázi použita kolekce

problems. Tyto definice mohou uživatelé využívat prostřednictvím webové aplikace. Správce se znalostí hesla může nové definice vytvářet a manipulovat s nimi. Každý dokument v této kolekci reprezentuje jedinečný problém, identifikovaný pomocí atributu **documentName** (názvu příkladu, který uloží správce).

7.2.1.1 Struktura dokumentu:

- **attributes**: Pole řetězců, kde každý řetězec představuje jeden atribut. Atributy jsou základními stavebními prvky v relačních schématech a slouží k definici struktury dat. (např. ["a", "b", "c", "d"]).
- **dependencies**: Pole objektů, kde každý objekt má strukturu:
    - **left**: Pole řetězců reprezentujících atributy na levé straně závislosti.
    - **righ**t: Pole řetězců reprezentujících atributy na pravé straně závislosti. (např. dependencies:[{left: ["a", "b", "c"]},right: ["d"]}, {left:["b","c"]},right: ["d", "a"]}])

7.2.2 Změna konfigurace

Změna konfigurace je jednoduchá – stačí upravit hodnoty v firebaseConfig objektu. Například, změna databaseURL na URL nově zřízené Firebase Realtime Database instance, která může být hostována v jiné geografické lokalitě nebo nakonfigurována s odlišnými parametry zabezpečení.

7.2.2.1 Příklady důvodů pro změnu konfigurace:

- **Zabezpečení**: V případě, že aplikace manipuluje s citlivými daty, může být vhodné provozovat databázi v regionu s přísnějšími zákony o ochraně dat nebo použít specifické konfigurace zabezpečení.
- **Náklady a správa zdrojů**: V závislosti na rozpočtu a požadavcích na úložiště může být výhodné přizpůsobit kapacitu databáze tak, aby odpovídala aktuálnímu využití bez nutnosti platit za nepoužívané zdroje.

7.3 Hlavní použité algoritmy v programu

7.3.1 Implementace a struktura tříd Algorithm.js a NormalFormALG.js

V rámci této bakalářské práce byly vyvinuty dva zásadní soubory, Algorithm.js a NormalFormALG.js, které představují jádro softwarového řešení pro vizualizaci

a manipulaci s algoritmy, používanými v relačních databázích. Tyto soubory jsou implementovány v jazyce JavaScript a jsou navrženy s důrazem na objektově orientovaný přístup, což zajišťuje modularitu a opětovnou použitelnost kódu.

7.3.2 Konstruktor třídy

Konstruktory tříd Algorithm i NormalFormALG inicializují všechny potřebné metody, které jsou vázány na instanci objektu. Tato inicializace zajišťuje, že všechny metody mohou být bez problémů použity v různých částech aplikace, což přináší vyšší úroveň modularizace a opětovné použitelnosti kódu. V konstruktoru jsou nastaveny vazby pro metody jako subset, intersection a další, které umožňují provádět základní i složitější operace nad množinami a funkčními závislostmi.

7.3.3 Použití tříd v projektu

Pro začlenění a využití tříd Algorithm a NormalFormALG v jakékoliv části projektu je nezbytné třídy naimportovat. Toto lze provést pomocí standardního importu v JavaScriptu, což umožňuje snadné použití všech metod třídy bez nutnosti opětovného definování funkčnosti. Import se obvykle realizuje na začátku souboru, kde je třída potřebná, což zvyšuje přehlednost kódu a usnadňuje jeho správu.

7.3.4 Výhody použití objektově orientovaného přístupu

Implementace tříd Algorithm a NormalFormALG v objektově orientovaném stylu přináší řadu výhod, včetně lepší struktury kódu, snížení redundance a zvýšení znovupoužitelnosti komponent. Tento přístup také usnadňuje údržbu aplikace a její další rozvoj, protože modifikace jedné metody neovlivní ostatní části programu, pokud nejsou přímo závislé na změněné funkcionalitě.

7.4 Možnosti rozšiřování aplikace

Pokud budete chtít přidat novou funkcionalitu do sekce “problém” tak doporučuji postupovat následovně:

1. Vytvořte nový adresář v cestě **problems/** a pojmenujte ho dle charakteristiky nové funkcionality, například NovyProblem.
2. V tomto adresáři vytvořte soubor JSX, který bude sloužit pro implementaci logiky a uživatelského rozhraní nové funkcionality. Současně vytvořte soubor SCSS pro přizpůsobení stylů.

3. Do souboru **problems/index.js** přidejte řádek exportu ve formátu: export { default as NovyProblem } from './NovyProblem'; kde NovyProblem je název vaší nové komponenty.
4. V **App.js** importujte novou komponentu z **./components/content/problems** a přidejte pro ni odpovídající směrovací cestu <Route>.
5. Do komponenty **problems/Problems.jsx** přidejte tlačítko, které umožní uživatelům přístup k nové funkci.
6. Zvažte použití existujících algoritmů z adresáře **algorithm/** pro podporu vaší nové funkcionality, případně vytvořte nové podle potřeb vaší aplikace.

   Pokud budete vytvářet nový algoritmus postupujte podle těchto dvou kroků:

   - Nejprve vytvořte novou metodu, která bude implementovat požadovaný algoritmus. Tento kód by měl být čistý a jasně komentovaný, aby bylo zřejmé, co funkce dělá a jaké parametry vyžaduje.
   - Přidejte tuto novou metodu do konstruktoru třídy a proveďte její vázání na instanci třídy. To zajistí správnou funkčnost klíčového slova this uvnitř metody, což je nezbytné pro správnou integraci a užití v rámci třídy.
7. Udržujte kód čistý a dobře okomentovaný. Všechny klíčové části kódu, zejména importy a závislosti, by měly být doplněny o komentáře vysvětlující jejich účel a funkci.

8 Závěr

V rámci bakalářské práce jsem vyvinul veřejně dostupnou webovou aplikaci, která řeší řadu nedostatků, které se nacházejí v podobných existujících řešeních. Zároveň zachovává všechny jejich klíčové funkce. Díky tomu, že aplikace byla vyvinuta později, mohla se těmito aplikacemi inspirovat, a zároveň využít nejnovější technologické inovace.

Oproti stávajícím řešením vidím hlavní výhodu v implementaci krokování jednotlivých postupů, vysvětlování jednotlivých kroků, a možnosti jejich vyzkoušení s okamžitou zpětnou vazbou.

Právě tyto vlastnosti mi během studia velmi chyběly. Ne vždy je z omezeného množství příkladů v přednáškách a na cvičeních pochopit, jak se určují redundantní atributy a redundantní závislosti. V mém případě mi s tímto pochopením pomohl spolužák, který mi znovu vysvětlil, že se to dělá pomocí atributového uzávěru a jak.

Stejně tak mne překvapovalo, že, ačkoli se finální řešení spolužáků od sebe lišila, vyučující je uznával všechny za správné. Až později jsem pochopil, že v mnoha krocích záleží na detailním pořadí atributů a závislostí, případně na jejich volbě v tom kterém okamžiku. Mně známé programy vracely pouze jeden jediný výsledek a proto jsem jej nemohl porovnat s jinou variantou, která mi vyšla. Stejně tak pro mne byl problematický algoritmus nalezení všech klíčů. Skutečné pochopení mi přineslo až to, když jsem algoritmus napsal, a na několika příkladech si ho odkrokoval.

Dle mého názoru bylo asi nejnáročnější. správné pochopení a vyřešení dekompozice. Nebyl jsem schopný pochopit, podle jaké závislosti mám relaci správně rozdělit, a po jejím rozdělení jsem leckdy nebyl schopen snadno určit, zda je relace v BCNF, ve 3NF, nebo zda je potřeba v dekompozici pokračovat. Navíc právě u tohoto algoritmu asi nejvíce záleží na volbě funkční závislosti, která bude v tom kterém kroku použita.

Právě z těchto důvodů jsem se v aplikaci zaměřil na co největší zdůraznění možnosti volby a dopadu takové volby na celkový výsledek. Jsem přesvědčen, že i díky tomu je výsledná aplikace pro výuku podstatně lepší a spolehlivější. Může pomoci studentům lépe se orientovat ve studijním materiálu a snadněji dosáhnout vyšší úrovně jejich odborné přípravy.

Seznam použité literatury

1. Studijní opory k přednáškám a cvičení z předmětu Databázové systémy od Ing. Lukáš OTTE, Ph.D. Vysoká škola báňská – Technická univerzita Ostrava, Fakulta strojní.
2. Pokorný J., Halaška, I: Databázové systémy. Skripta, 2. přepracované vydání, Vydavatelství ČVUT, 2003.
3. Slidy předmětu NDBI025 - Databázové systémy od doc. RNDr. Tomáš Skopal, Ph.D. RNDr. Michal Kopecký, Ph.D.
4. Zbyněk Bureš DATABÁZOVÉ SYSTÉMY 1 1. vydání ISBN 978-80-87035-88-7 Vydala Vysoká škola polytechnická Jihlava, Tolstého 16, Jihlava, 2014. Dostupné z: https://kdep.vse.cz/wp-content/uploads/page/186/DEP509_Databazove-systemy_Zdenek-Bures.pdf
5. Bakalářský projekt Dany Soukupové (vedl dr. Říha). Dostupné z: https://www.ms.mff.cuni.cz/~kopecky/vyuka/ndbi025/siret.ms.mff.cuni.cz/skopal/databaze/DatAlg.zip
6. Normalizace databáze. Dostupné z: https://https://cs.wikipedia.org/wiki/Normalizace_datab%C3%A1ze citováno 17. 4. 2023

Seznam tabulek

Seznam použitých zkratek

- 1NF: 1. normální forma
- 2NF: 2. normální forma
- 3NF: 3. normální forma
- BCNF: Boyce-Coddova normální forma
- FZ: Funkční závislost

Seznam obrázků