ZÁPADOČESKÁ UNIVERZITA V PLZNI

FAKULTA EKONOMICKÁ

Diplomová práce

Návrh a implementace výukové hry typu zebra

Creation of Einstein's puzzle educational game

Vít Černohouz

Plzeň 2012

Prohlášení

Prohlašuji, že jsem diplomovou práci na téma

„Návrh a implementace výukové hry typu zebra“

vypracoval samostatně pod odborným dohledem vedoucího diplomové práce za použití pramenů uvedených v přiložené bibliografii.

V Plzni, dne 23. 4. 2012 ……

podpis autora

Poděkování

Chtěl bych poděkovat RNDr. Mikuláši Gangurovi Ph.D. za odborné vedení při diplomové práci. Dále bych chtěl poděkovat svým rodičům za psychickou podporu při psaní práce.

Obsah

Úvod

Autor této práce věří, že každý z Vás, kdo čte tuto práci, se v minulosti pokusil vyřešit nějakou logickou úlohu. Jednou ze známých logických úloh je také úloha typu zebra[1].

V této úloze se snažíme zjistit jednoznačná přiřazení mezi členy různých entit. Za entity můžeme považovat například lidi, jejich oblíbené jídlo, značku jejich auta a podobně. Řešením úlohy je například zjištění, že Jiří má rád ryby a značka jeho auta je Škoda a že Petr má rád hovězí a značka jeho auta je Volvo. K vyřešení úlohy nám slouží indicie, jejichž počet ovlivňuje obtížnost. Tyto indicie se dají vyjádřit větami a tím, že hráč zohlední všechny nebo jen některé z nich a uvažuje nad jejich logickými vazbami, tak se postupně odhalí příslušné výsledné vztahy.

Řešením logických úloh typu zebra si člověk může ve volném čase například zlepšit svoji duševní kondici nebo by se mohl za pomoci řešení těchto úloh připravovat přijímací zkoušky na řadu vysokých škol, kde se tento typ úloh vyskytuje.

V současnosti existuje několik typů úlohy zebra, které se různí svoji formou. Může jít o hry na mobilních telefonech nebo na počítačích anebo se úlohy vyskytují pouze v psané formě například v časopisech mezi křížovkami, kdy v příštím čísle je uveřejněno řešení. Obsahově se úlohy dají rozdělit podle složitosti, délky a podle různých způsobů zadání.

Cíl práce:

Hlavním cílem práce je návrh a naprogramování generátoru zadání pro hru typu zebra. Práce se zaobírá různými přístupy pro řešení úlohy a posléze zvolíme nejlepší způsob vhodný pro implementaci programu. Teoretická východiska práce jsou matematická logika a teorie grafů. V těchto oblastech bude provedena rešerše.

[1] V angličtině se používá název Einstein's puzzle podle logické úlohy, kterou vymyslel v první polovině 20 století Albert Einstein. Tento název se používá také v češtině jako Einsteinova hádanka.

1. Základy logiky

1.1. Výroková logika

Jestliže máme logicky zkoumat nějaký text, například právní, rozebereme ho na elementární části a hledáme mezi nimi všechny možné vztahy. Touto analýzou se snažíme docílit získání nového tvrzení neboli dodatečné informace, která dříve nebyla patrná. Základem teorie moderní logiky a možných logických analýz textů je výroková logika. Elementární částí, o které jsme mluvili, je *výrok*. Je to určité tvrzení, které může být pravdivé (budeme značit 1) nebo nepravdivé (budeme značit 0). Jednoduché výroky se dají sestavit do výroků složených, kde spojujeme jednoduché výroky pomocí logických spojek. Tyto spojky určují jednoznačný logický vztah mezi výroky. Každý složený výrok má také svojí výslednou pravdivostní hodnotu. (Kotlán, 2006)

Níže uvádíme logické spojky a jim příslušné používané znaky.

- *a, a zároveň* $\wedge$
- *nebo* $\vee$
- *Jestliže…, pak* $\rightarrow$
- *právě tehdy, když* $\leftrightarrow$

K výčtu logických operátorů je také vhodné přiřadit operátor negace, tedy tvrzení, že něco není pravdivé.

- *není pravda, že* $\neg$

Označme jednotlivé výroky velkými písmeny A, B. Při různých logických vazbách mezi výroky dostaneme složený výrok s danou výslednou logickou hodnotou, tedy *pravda = 1*, *nepravda = 0*. Složené výroky s ohledem na logické spojky rozlišují na konjunkci, disjunkci, implikaci a ekvivalenci.

- Výrok A a zároveň výrok B je pravdivý, právě když jsou oba výroky pravdivé. Jedná se o konjunkci.
- Jestliže je výrok A pravdivý nebo výrok B pravdivý, pak je složený výrok pravdivý. Spojka nebo značí disjunkci.

- Druhý typ disjunkce je vylučující disjunkce neboli alternativa, kdy je složený výrok pravdivý, když platí právě jeden výrok.
- Implikace (Jestliže platí A, pak platí B) je nepravdivá pouze v případě, kdy A je pravda a B je nepravda.
- Ekvivalence (Platí A právě tehdy, když platí B) je pravdivá, pokud platí zároveň A i B nebo neplatí zároveň A i B.

Tato pravidla shrneme do přehledné tabulky s jednotlivými logickými operátory v záhlaví. Postupně jde zleva o pravdivostní hodnotu výroku A, pravdivostní hodnotu výroku B, konjunkci, disjunkci, alternativu, implikaci a ekvivalenci.

Tabulka 1: Pravdivostní tabulka pro logické operátory

A	B	$A \wedge B$	$A \vee(1) B$	$A \vee(2) B$	$A \rightarrow B$	$A \leftrightarrow B$
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	0
1	1	1	1	0	1	1

Nyní je třeba také upřesnit negaci výroku. Mějme výrok

- C = „Jídlo je dobré.“

Jestliže chceme výrok znegovat, tedy dostat jeho logický opak, můžeme výrok uvést větou: „Není pravda, že…“. Jestli chceme ale výrok C přizpůsobit pouze změnou slov, tak je třeba znegovat pouze jedno klíčové slovo a to buďto sloveso nebo přídavné jméno. Dostaneme tak následující znegovaná tvrzení:

- $\neg$ C = „Jídlo není dobré.“
- $\neg$ C = „Jídlo je špatné.“

Nesmíme znegovat obě slova najednou. Jednalo by se o dvojitou negaci, jejímž výsledkem je původní výrok.

- C´ = „Jídlo není špatné.“

Když porovnáme výrok C a C´, tak z češtinářského hlediska dostaneme poněkud jiný význam. Ve formální logice jsou ale slova dobrý a špatný opakem a tak se výroky C a C´ shodují.

Důležité je také objasnit použití negace v případě slov některý a všichni. V logice se tato slova nazývají kvantifikátory. A to obecný a existenční kvantifikátor. Obecný i existenční kvantifikátor jsou vždy na začátku výroku. Obecný kvantifikátor můžou zastupovat slova všichni, každý, pro všechny a podobné. Naproti tomu existenční kvantifikátor se může skrývat za slovy některý, někdo, jeden.

- obecný kvantifikátor ∀
- existenční kvantifikátor ∃

Mějme tedy následující výrok.

- D = „Všichni žáci čtou rádi fantasy.“

V tom to výroku D máme obsažen obecný kvantifikátor „Všichni“. Když bychom znegovali pouze sloveso „čtou“ na „nečtou“, nedostali bychom správnou negaci výroku. Za negaci nelze ani považovat znegování „Všichni“ na „Nikdo“. Správný postup je změnit kvantifikátor z obecného na existenční a znegovat klíčové slovo. Níže je tedy uvedena správná negace.

- ¬ D = „Někteří žáci nečtou rádi fantasy.“

Tím, že již existuje alespoň jeden žák, který nečte rád fantasy, tak tím naruší a úspěšně zneguje celý původní výrok „Všichni žáci čtou rádi fantasy.“

Dále bude vysvětlen pojem premisa a závěr. Premisa je takový výrok, který je daný jako předpoklad na počátku úlohy a je pravdivý. Logickým postupem lze z množiny premis určit pravdivost závěru. Příklad můžeme uvést následující.

Zjistěte na základě následujících premis pravdivost závěru. *Když bude venku teplo, půjdu se koupat. Venku není teplo. Závěr: Půjdu se koupat.*

- Premisa A: *Když bude venku teplo, půjdu se koupat.*
- Premisa B: *Venku není teplo.*
- Závěr: *Půjdu se koupat*

Premisa A je zde složený výrok a premisa B je jednoduchý výrok a závěr je také jednoduchý výrok.

Výsledek u tohoto výroku je naše tvrzení, jestli je závěrečný výrok pravdivý. U tohoto jednoduchého příkladu je správné tvrzení, že je závěr nepravdivý.

U složitějších úloh by se mohla použít tabulková metoda, kde složitost úlohy roste exponenciálně s počtem jednoduchých výroků obsažených v premisách. Tedy jestliže je počet premis *n,* tak složitost je následující. (Kotlán, 2006)

$$O = 2^n$$

Při řešení úlohy jde přímo o počet řádků v pravdivostní tabulce používané pro výpočet.

1.2. Predikátová logika

Další známá oblast z logiky je logika predikátová. Tato oblast se zabývá soudy a vztahy mezi nimi, kdy se každý soud skládá ze *subjektu*, *predikátu* a *spony*. (Kotlán, 2006)

Důležité jsou zejména tyto pojmy.

- Subjekt - Označuje určité individuum, věc neboli prostě subjekt, který právě zkoumáme. Například by mohlo jít o konkrétní osoby, zvířata, auta, města, rostliny a podobné. Subjekty ve formálním predikátovém jazyce označujeme malými písmeny. Například subjekt *Adam* můžeme nahradit písmenem *a* a subjekt *Petr* písmenem *b*.
- Predikát – představuje určité vlastnosti, atributy subjektu. Může jít o několik slov, která se vždy vztahují k některému ze subjektů. Některé uvedeme jako příklad.
    - *Je milý.*
    - *Umí plavat.*
    - *Jezdí na koni*
    - *Pije pivo.*
    - *Je lyžař.*
    - *Nekouří*

Pro formální jazyk predikátového jazyka se používá také dříve zmíněných obecných a existenčních kvantifikátorů.

Běžné věty můžeme přepisovat do jazyka predikátové logiky, čímž docílíme formálnějšího a přehlednějšího zobrazení vět a jejich významu a také nám bude umožněno na ně aplikovat algoritmické postupy jako například rezoluční metodu, která

nám zjišťuje pravdivost určitého přiřazení predikátu a subjektu nebo přímo zjišťuje, který subjekt náleží danému predikátu. (Kotlán, 2006)

Níže uvedeme příklad převedení věty do predikátového jazyka.

Převeďte následující větu do jazyka predikátové logiky: *„Někteří lidé jsou pracovití, a přesto neočekávají odměnu.“*

- Slova *„Někteří lidé“* převedeme následovně. *„Lidé“* označíme *l* a „někteří“ nám oznamuje, že na *l* bude použit existenční kvantifikátor (znak ∃).
- Spojka „*a přesto“* nám zastupuje konjunkci (znaménko ∧).
- Dále identifikujeme a substituujeme predikáty. „*Být pracovitý“* označíme velkým písmenem *P* a „Očekávat odměnu“ označíme *O*.
- Když správně složíme dohromady všechny symboly, dostaneme následující výraz predikátového jazyka.

$$\exists\, l\, \{P\,(l) \wedge \neg O\,(l)\}$$

Vidíme, že predikáty označené velkými písmeny mají jako argument daný subjekt *Lidé*. Tímto zápisem se jazyk zpřehlední v případě, že zkoumáme nebo převádíme složitější výrazy.

2. Základy z teorie grafů

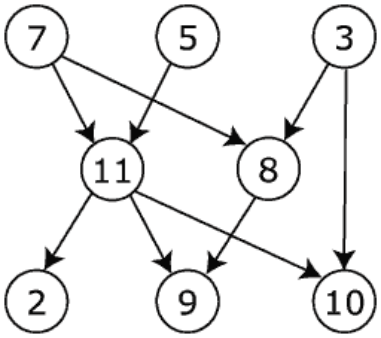
V práci budeme pro reprezentaci a lepší orientaci v problematice úlohy používat grafovou strukturu zvážíme použití některých postupů z teorie grafů.

2.1. Základní pojmy

Hlavními pojmy v teorii grafů jsou *graf*, *uzel* a *hrana*.

- *Uzel* si můžeme představit jako jednoduchý bod obsahující určitou informaci (číslo, řetězec apod.). Uzel značíme u_i. Kde *i* je přirozené číslo a znamená očíslování uzlu.
- *Hrana* spojuje vždy dva uzly. Uzly spojené hranou se nazývají *sousedící uzly*. Hrana může být orientovaná a neorientovaná. Orientovaná hrana vyjadřuje jednosměrný vztah od jednoho uzlu k druhému a neorientovaná vyjadřuje vztah obousměrný. Značení pro hranu je $h\ (u1,\ u_2)$. Pro orientovanou hranu se $h\ (u1,\ u_2)$ nerovná $h\ (u_2,\ u_1)$. Pro neorientovanou jsou zápisy shodné.
- *Graf* je množina uzlů a hran. Jestliže má všechny hrany neorientované, pak se jedná o graf neorientovaný. V opačném případě jde o orientovaný graf.

Obrázek 1: Příklad orientovaného grafu (Zdroj: vlastní)



2.2. Matice sousednosti

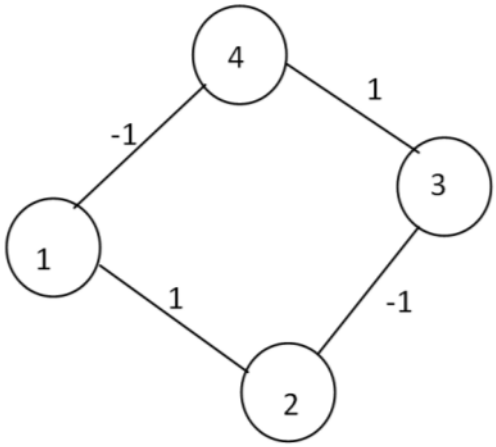
Matice sousednosti je převedení grafu na matici. Každý vrchol je očíslován přirozeným číslem *1, 2, ..., n*, kde *n* je počet vrcholů a vytvoříme matici A *n* x *n*. jestliže vede hrana z vrcholu *i* do vrcholu *j*, kde $i,j \in \langle 1, 2, ..., n \rangle$, tak v matici hodnotu a_{ij} nastavíme na 1. Hodnoty matice A reprezentují všechny možné hrany vedoucí mezi vrcholy. Pro hrany spojující vrcholy je hodnota v matici rovna 1, ostatní hodnoty matice, tedy neexistující hrany, jsou rovny 0. (Ryjáček, 2007)

Je také možné použít nulu v případě, že mezi vrcholy není hrana a ohodnocení hrany v případě, že mezi vrcholy je hrana, která má přiřazenou číselnou hodnotu.

V případě, že se jedná o neorientovaný graf, je matice sousednosti symetrická podle hlavní diagonály.

Například můžeme mít následující graf se čtyřmi vrcholy. Můžete jej vidět na následujícím obrázku. Mezi vybranými vrcholy jsou neorientované ohodnocené hrany.

Obrázek 2: Graf čtyř vrcholů pro vytvoření matice sousednosti (Zdroj: vlastní)



Tento graf na obrázku 2 nyní vyjádříme pomocí matice sousednosti.

Tabulka 2: Matice sousednosti (Zdroj: vlastní)

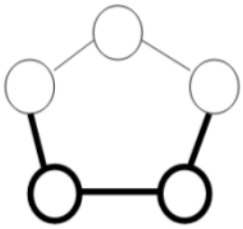
	1	**2**	**3**	**4**
1	0	1	0	-1
2	1	0	-1	0
3	0	-1	0	1
4	-1	0	1	0

V této tabulce je vidět matice sousednosti neorientovaného grafu s ohodnocenými hranami. První sloupec a řádek tabulky obsahuje tučně vypsaná očíslování vrcholů a hodnoty ohodnocení hran jsou umístěny za těmito sloupci.

2.3. Podgraf

Jestliže máme graf *G*, tak podgraf grafu *G* je graf obsahující stejně nebo méně hran a vrcholů jako graf *G*. Podgraf můžeme označit jako *G´*. Symbolicky je podgraf znázorněn takto: $G' \subset G$ (Ryjáček, 2007)

Obrázek 3: Podgraf (Zdroj: vlastní)



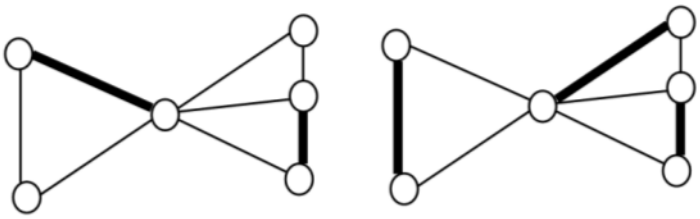
2.4. Párování a bipartitní graf

V této podkapitole budeme pojem graf považovat vždy jako neorientovaný.

Párování je takový podgraf grafu G, že obsahuje některé dvojice uzlů spojené hranou, přičemž žádné z těchto dvojic mezi sebou nejsou spojeny hranou. (Ryjáček, 2007)

Má smysl hledat *maximální* a *největší* množinu *párování*. Největší párování je takové maximální párování, které obsahuje nejvíce uzlů.

Obrázek 4: Maximální a největší párování v grafech (Zdroj: Ryjáček, 2007)

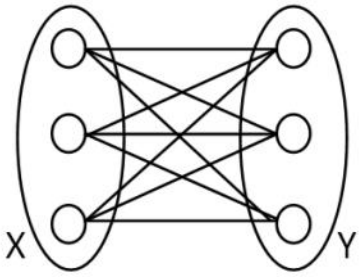


Tyto dva grafy mají oba maximální párování. Největší párování má ale pouze graf napravo. Tento graf má zároveň *perfektní párování*, což znamená, že jsou v podgrafu párování obsaženy všechny uzly původního grafu. Je zřejmé, že nutná podmínka existence perfektního párování je sudý počet uzlů. (Ryjáček, 2007)

Bipartitní graf neboli *bigraf* je definován takto:

Bigraf je takový graf G, jehož množinu uzlů lze rozdělit na neprázdné disjunktní podmnožiny X, Y tak, že pro každou hranu *(u, v)* z množiny hran grafu G platí, že *u* je vrchol množiny *X*, *v* je vrchol množiny *Y*. (Ryjáček, 2007)

Obrázek 5: Bipartititní graf (Zdroj: vlastní)



Příklad úlohy nalezení největšího párování:

Máme skupinu chlapců X a děvčat Y. Tyto množiny představují vrcholy grafu. Hranami spojíme každého chlapce s děvčetem, když se znají. Potom hledáme největší párování, abychom vytvořili taneční páry.

Bigraf má vždy sudý počet vrcholů a tak je zajištěna nutná podmínka existence perfektního párování

Postačující podmínku existence párování celé množiny X v bigrafu nám zajistí Hallova věta. Až budeme používat tuto podmínku, budeme pracovat s bigrafem, který má $X = Y$, tak bude Hallova věta dokonce postačující pro získání perfektního párování.

Hallova věta:

Bipartitní graf G = (X, Y) má párování pokrývající všechny uzly z X, právě když pro každou množinu uzlů S podmnožina X platí $|N(S)| \geq |S|$. (Ryjáček, 2007)

kde

- *N (S) jsou sousedící vrcholy množiny S.*

- $|S|$ *je počet vrcholů množiny S*

Jestliže tedy chceme, aby v bigrafu se stejným počtem vrcholů v množině X a v množině Y existovalo perfektní párování, tak je nutné, aby každá podmnožina měla menší nebo stejný počet vrcholů než počet sousedních vrcholů v Y.

3. Logická hra zebra

3.1. Popis hry

Značení v tabulce

V následující tabulce rozlišujeme tři hodnoty:

- 0 – Nevíme, jestli mezi objekty je nebo není relace.
- 1 – Mezi objektem v řádku a ve sloupci je relace.
- -1 – Mezi objektem v řádku a ve sloupci není relace[2].

Tabulka 3: Jednoduchý příklad hry zebra se dvěma kategoriemi po třech objektech

		Osoba			Obydlí		
		Pavla	Šimon	Eliška	Byt	Chata	Vila
Osoba	Pavla	/	/	/	-1	1	-1
	Šimon	/	/	/	-1	-1	1
	Eliška	/	/	/	1	-1	-1
Obydlí	Byt	-1	-1	1	/	/	/
	Chata	1	-1	-1	/	/	/
	Vila	-1	1	-1	/	/	/

Jedná se o úlohu typu zebra, jejíž generování zadání je předmětem této práce. Hru lze zařadit do zábavných logických her pro jednoho hráče, kdy pro vyřešení stačí většinou jen tužka a papír. Jde o určitý hlavolam, kde se hráč na základě daného zadání snaží nalézt vztahy mezi objekty z různých skupin.

V tabulce 3 můžeme vidět jednoduchý příklad vyřešené úlohy zebra pro dvě kategorie, přičemž v každé kategorii jsou tři objekty. Toto zobrazení úlohy v podobě tabulky

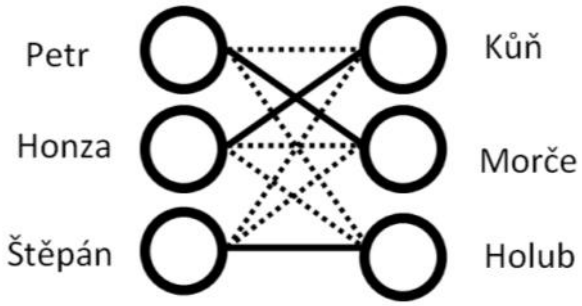
[2] Podrobnější vysvětlení těchto vztahů bude podáno níže.

funguje jako matice sousednosti zmíněné v předcházející kapitole. Objekty jsou jednotlivé vrcholy grafu (přičemž neuvažujeme smyčky) a vztahy mezi objekty představují hrany.

Řešení úlohy a zadání

Řešením úlohy rozumíme existencí množiny přiřazení každého objektu z jedné kategorie k různým objektům z jiné kategorie vyhovující zadané množině predikátů, ke které se dostaneme později. V řeči teorie grafů je řešením úlohy nalezení perfektního párování mezi jednotlivými objekty kategorií.

Obrázek 6: Jedno z řešení zobrazeno jako bipartitní graf



Množina predikátů neboli indicií[3] v zadání je vlastně částí řešení úlohy a tím, že lze z této množiny odvozovat další informace (další predikáty), pak, jestliže počáteční množina indicií obsahuje dostatek klíčových indicií, je možné nalézt postupně tolik informací, že se nám odhalí celé řešení úlohy.

Zadání bude v této práci definováno jako minimální množina predikátů, což je množina několika predikátů, z nichž je možné odvodit řešení úlohy a zároveň platí, že jestliže odebereme jakýkoliv predikát, již řešení s danou množinou nemůžeme nalézt. Postup nalezení této minimální množiny predikátů bude popsán v kapitole 3.

[3] Slovo *indicie* bylo použito jako česká alternativa k překladu anglického slova *clue*, které se dá také přeložit jako vodítko, stopa nebo nápověda, tedy informace, která vede k objasnění nebo částečnému objasnění určitého problému či úlohy. Potom množina indicií je určitý souhrn několika nápověd, který nám částečně nepřímo objasňuje řešení úlohy. Tento termín je použit zejména v programové části řešení a v této práci je ekvivalentní s pojmem predikát.

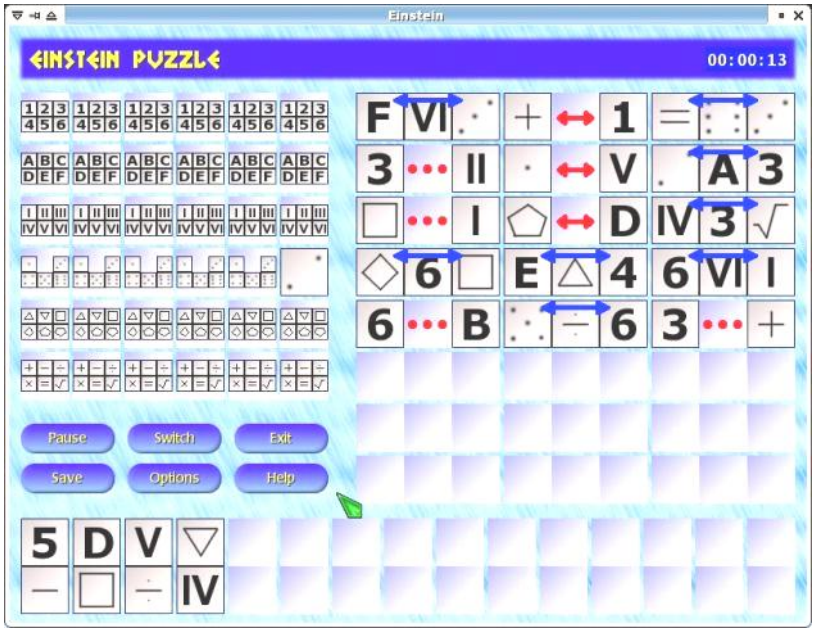
Predikát a indicie

Pro potřeby této práce si definujeme indicii pro hru zebra jako predikát, který určuje vlastnost jednoho objektu týkající se druhého objektu. je to tedy predikát se dvěma parametry, což jsou různé objekty.

Například: ¬má (Petr, pes) = *„Petr nemá psa“*

Ať bude v dalších částech práce uveden pojem predikát nebo indicie, bude znamenat tento popsaný druh predikátu.

Obrázek 7: Grafická verze hry zebra (Zdroj: http://linux.softpedia.com)



Zadání hry bývá obyčejně napsáno jako text, který nás uvede do situace, seznámí nás s jednotlivými objekty a vztahy s ostatními objekty.

Například: *„V jedné ulici vedle sebe žili tři sousedi. Každý měl své obydlí. Tito lidé se jmenovali Pavla, Šimon a Eliška. Bydleli v bytě, chatě a vile. Nevíme však, kdo bydlel v kterém.“* … Dále by následovaly indicie, které by postupně objasňovaly řešení, tedy v tomto případě, kdo kde bydlí.

Například uvedeme několik indicií: „Eliška nebydlí ve vile“, „Pavla nebydlí vedle Bytu“, „Šimon bydlí nalevo od Vily“.

Jiné druhy tohoto hlavolamu bývají znázorněny graficky (Obrázek 7: Grafická verze hry zebraObrázek 7) pouze jako symboly, které jsou úmyslně umístěny v sloupcích nebo řádcích. Pro řešení obou typů úloh je vhodné si vytvořit na papír tabulku, kde si budeme přehledně označovat vazby mezi objekty. Taková tabulka bude uvedena dále (Tabulka 4). Když bychom převedli tato zadání do obecné formy, mohli bychom jej popsat takto.

- **Kategorie *k***
  - Jednotlivé kategorie (jako například osoby, zvířata, domy, značky aut, obchody) můžeme označit přirozenými čísly. Označíme je písmenem *k*.

$$k = 1, 2, \ldots n$$

  - kde *n* je počet kategorií.

- **Množina objektů *O***
  - Jde o konkrétní názvy nebo jména osob, zvířat a ostatních věcí z reálného světa. Například: „Karel“, „Jan“, „pes“, „morče“, „džus“, „pivo“ a další. Tyto objekty náleží vždy určité kategorie. Zde pro kategorii *1, 2, ... n* označíme objekty následovně.

$$O_{1k}, O_{2k}, \ldots O_{mk} \qquad \text{pro } k = 1, 2, \ldots n$$

  - Kde *k* je kategorie.
  - *m* je počet objektů v jedné kategorii.
  - počet všech objektů označíme *w* a platí.

$$w = k * n$$

  - Zkráceně zapíšeme všechny objekty takto:

$$O_{lk} \quad \text{, kde } l = 1, 2, \ldots m.$$

  - O_{12} například značí první objekt (l = 1) náležící druhé kategorii (k = 2).
  - Množina všech objektů, kterou označíme *O*, bude vypadat takto.

$$O = \{ O_{lk} \} \quad l = 1, 2, \ldots m \text{ pro } k = 1, 2, \ldots n$$

- Množina všech objektů připadající první kategorii, zapíšeme následovně.

$$O_1 = \{ O_{lk} \} \quad l = 1, 2, \ldots m \text{ pro } k = 1$$

- Schéma *S* – schéma je daná množina objektů patřících k jejím kategoriím. Například*: „Máme tři obydlí a tři osoby. Osoby jsou Eliška, Pavla a Šimon a obydlí jsou chata, byt a vila.“* Toto je jedno možné schéma. Podle tohoto schématu se bude řídit vytváření zadání příkladu.
- Množina všech řešení R schématu S – Jedno dané schéma, které jsme jako příklad zmínili v předcházející odrážce, může mít více řešení. Jedno řešení můžeme nalézt v tabulce 4 a snadno si můžeme vymyslet další. Například: {*„Eliška má vilu“; „Pavla má byt“; „Šimon má chatu“} nebo {„Pavla má vilu“; „Eliška má byt“; „Šimon má chatu“} ... E*xistují další možná přiřazení objektů, tedy řešení. Množinu všech řešení *R (*schématu *S)* označíme *R(S). R(S)* obsahuje všechna možná řešení :

$$r_1(S), r_2(S), \ldots r_p(S)$$

kde r_j je některé konkrétní zadání z *R(S),*

$$j \in \langle 1, 2, \ldots, p \rangle ,$$

kde *p* je počet řešení daného schématu *S,*

Číslo *p* roste přímo úměrně s *m* a *n*.

- Jedno řešení schématu *S* se značí $r_j(S)$. Například jde o konkrétní řešení daného schématu *S*:

  {„Eliška má vilu“; „Pavla má byt“; „Šimon má chatu“}

  - Objekty jsou v tomto případě pro kategorii Osoba (k = 1) tyto:
    - O_{11} = Pavla
    - O_{21} = Šimon
    - O_{31} = Eliška
  - A objekty pro kategorii Obydlí (k = 2) tyto:
    - O_{12} = Byt
    - O_{22} = Chata
    - O_{32} = Vila
  - A vztahy mezi objekty označíme pomocí tří typů predikátů:

- **má**[4] (Objekt A, Objekt B) – například má (O_{31} a O_{32}) je ekvivalentní s tvrzením: *„Eliška má vilu“*. V tabulce budeme tento stav značit jedničkou (**1**).
- **¬má** (Objekt A, Objekt B) - například ¬má (O_{31} a O_{12}) je ekvivalentní s tvrzením plynoucím z předcházejícího příkladu: *„Eliška nemá byt“*. V tabulce budeme tento stav značit mínus jedničkou (**-1**).
- **může_mít** (Objekt A, Objekt B) – například, kdybychom ještě neznali povahu vztahu mezi objekty, tak jsou ekvivalentní tato dvě vyjádření: může_mít (O_{31} a O_{32}) a *„Nevíme, jestli Eliška má vilu“*. Tento stav v tabulce značíme nulou (**0**). Toto vyjádření se v případě $r_j(S)$ nevyskytuje, protože všechny tři osoby mají přiřazené obydlí (třikrát má) a tím pádem jsou všechny ostatní relace s osobami typu relace_ne (šestkrát). Všech devět možností relací je vyplněno jasným vztahem *ne* nebo *ano* a není zde místo pro žádné alternativy (relace_nevím).

  - Toto řešení, *{„Eliška má vilu“; „Pavla má byt“; „Šimon má chatu“},* tedy můžeme formálně zapsat takto.
    - *má* (O_{31}, O_{32})
    - *má* (O_{11}, O_{12})
    - *má* (O_{21}, O_{22})
    - Pomocí těchto tří relací máme definované jedno řešení daného schématu *S*.

- Množina predikátů jednoho řešení (označíme $P(r_j(S))$
  - Množina všech predikátů – $P(r_j(S))\_all$[5] – tato množina obsahuje všechny možné predikáty k jednomu danému řešení schématu *S*. Podle minulého příkladu –

[4] Predikáty jsou v tomto případě „symetrické“. Tedy platí, že má (O_{11}, O_{12}) = má (O_{12}, O_{11}). Například tvrzení „Eliška má chatu.“, nám poskytne i informaci, že „Chata má Elišku.“.

[5] Dále budeme pro množinu všech predikátů pro dané řešení *r* používat zkrácené značení *P(r)*.

{„Eliška má vilu“; „Pavla má byt“; „Šimon má chatu“}

- O_{11} = Pavla
- O_{21} = Šimon
- O_{31} = Eliška
- O_{12} = Byt
- O_{22} = Chata
- O_{32} = Vila

– bude za předpokladu dvou typů predikátů (má, ¬má) množina všech predikátů takováto.

1. *má (O_{31}, O_{32})*
2. *má (O_{11}, O_{12})*
3. *má (O_{21}, O_{22})*
4. *¬má (O_{31}, O_{12})*
5. *¬má (O_{31}, O_{22})*
6. *¬má (O_{11}, O_{22})*
7. *¬má (O_{11}, O_{32})*
8. *¬má (O_{21}, O_{12})*
9. *¬má (O_{21}, O_{32})*

- Minimální množina predikátů *$P(r_j(S))\_min$* – tato množina obsahuje pouze omezený počet predikátů, který dostačuje k vyřešení úlohy. Tedy taková množina, která nám poskytne dostatek informací, vedoucích (při dodržení správného postupu) k vyřešení úlohy. A také platí, že kdybychom odstranili jediný, jakýkoliv predikát z této množiny, tak by již nebylo možné nalézt řešení. Příklad takovéto minimální množiny je uveden níže. Tato množina nám stačí k vyřešení úlohy. První predikát nám dává tuto informaci: *„Eliška má vilu.“* a druhý predikát znamená *„Šimon nemá byt.“* Šimon tedy nemůže mít byt a ani vilu. Zbývá tedy jenom byt, tak zjišťujeme platnost dalšího predikátu z množiny všech predikátů *$P(r_j(S))\_all$: „Šimon má chatu“* a zbyla jen Pavla a byt, tak určíme i poslední platný predikát: *„Pavla má byt“*.

1. *má* (O_{31}, O_{32})
2. ¬*má* (O_{21}, O_{12})

Různá zadání mohou hráči působit větší nebo menší námahu. Záleží na různých faktorech.

Například může hráči napomoci přehledná volba kategorií. Potom si snáze vytvoří jasnější představu o celé hře. Přehledné kategorie můžou být například: lidi, jejich mazlíčci, auta, oblíbený nápoj a podobně. Kdybychom místo názvů kategorií a objektů použili například pouze písmena nebo čísla, úloha by se stala hůře řešitelnou až matoucí a navíc i méně poutavou, což by jistě nebyl náš cíl. Pro počítač by byla tato možnost ztížení úlohy zbytečná, ale schopnost vizuální představivosti člověka by byla velmi omezena a tak by se zpomalilo řešení celé úlohy.

Jestliže by množina predikátů *P* na počátku úlohy obsahovala nadbytečné predikáty, tak by byla úloha také usnadněna. Hráč by nemusel provádět tolik zjišťování a provádět tolik operací k vyřešení úlohy. Je důležité najít způsob, jak dosáhnout nalezení minimální množiny *P*, která nám zaručuje objasnění všech vztahů mezi objekty.

Dalším možným ztížením je vynucení řešení úlohy pomocí složitějších vztahů. Toho je možné dosáhnout například pomocí tranzitivních vztahů, které budou definovány v další podkapitole. Tato záležitost se také týká způsobu generování množiny *P*.

3.2. Jednoduchý příklad hlavolamu

Tabulka 4: Příklad úlohy se 2 kategoriemi, každá se 3 objekty (Zdroj: vlastní)

		Osoba			Zvíře		
		Petr	Jana	Tomáš	Pes	Kočka	Morče
Osoba	Petr	/	/	/	1	-1	-1
	Jana	/	/	/	-1	1	-1
	Tomáš	/	/	/	-1	-1	1
Zvíře	Pes	1	-1	-1	/	/	/
	Kočka	-1	1	-1	/	/	/
	Morče	-1	-1	1	/	/	/

Toto je jednoduché řešení úlohy zebra se dvěma kategoriemi se třemi objekty, kde hráč po vyřešení přiřadil všechny správné vztahy a tak zjistil, že

- Petr má psa.
- Jana má kočku.
- Tomáš má morče.

Tato jedna úloha má více možných minimálních množin predikátů (zadání), která nás tedy dovedou ke stejnému řešení. Dále je pro tyto kategorie (osoby a zvířata) a objekty (Petr, Jana, Tomáš a pes, kočka, morče) možné sestavit více řešení.

Například:

- Petr má morče.
- Jana má kočku.
- Tomáš má psa.

Nebo:

- Petr má kočku.
- Jana má psa

- Tomáš má morče.

Každé z těchto a ostatních možných řešení má svojí množinu různých zadání. Tento fakt bude v příštích kapitolách důležitý pro generování zadání pro danou úlohu.

3.3. Postupy řešící úlohu

Mějme takovou množinu predikátů, která není dostačující pro dosažení řešení úlohy. Jedině jejím rozšířením o nové predikáty lze nalézt všechny neznámé vztahy mezi objekty a docílit tak řešení. Každé rozšíření úlohy o další predikát lze provést jedním z následujících způsobů. Připomeňme, že množina predikátů se skládá ze vztahů mezi objekty, a to buď ano (predikát má), nebo ne (predikát ¬má). Totiž i informace, že nejsou ve vztahu, nám může pomoci nalézt řešení.

V následujících postupech je vždy uveden případ, kdy je možné použít daný postup. Je popsáno použití daného postupu a jako názorná grafická ukázka je ke každému postupu připojena tabulka, kde jsou klíčové hodnoty buněk vypsány. Tučně vypsané hodnoty buněk jsou ty, ke kterým se daným postupem dospělo.

1. **Dva objekty jsou ve vztahu (predikát má) - vyvozením**
    - Objekt O_{11} není ve vztahu má s žádným objektem než O_{22}, pak musí platit *má* (O_{11}, O_{22}).
    - Příklad: Eliška bydlí buď v bytě, na chatě, nebo ve vile. Dále víme, že Eliška nebydlí ani na chatě, ani v bytě. Musí tedy nutně bydlet ve vile.
        - vstup:
        - *relace_ne* (O_{11}, O_{12}) = *„Eliška nebydlí v bytě“*
        - *relace_ne* (O_{11}, O_{32}) = *„Eliška nebydlí na chatě.“*
        - výsledek:
        - *má* (O_{11}, O_{22}) = *„Eliška bydlí ve vile.“*

Tabulka 5: Demonstrace získání predikátu má - vyvozením (Zdroj: vlastní)

		Osoba			Obydlí		
		Eliška	Pavla	Šimon	Chata	Vila	Byt
Osoba	Eliška	/	/	/	-1	***1***	-1
	Pavla	/	/	/	0	0	0
	Šimon	/	/	/	0	0	0
Obydlí	Chata				/	/	/
	Vila				/	/	/
	Byt				/	/	/

2. Objekty nejsou ve vztahu (predikát ¬má) - vyloučením

- Když máme *má* (O_{11}, O_{21}). Potom O_{11} není ve vztahu s ostatními objekty z kategorie 2 a O_{21} není ve vztahu s ostatními objekty z kategorie 1. Toto zjištění je předvedeno v příkladě níže.
- Příklad: Eliška může bydlet v bytě, na chatě nebo ve vile. Ve vile může bydlet Pavla, Eliška nebo Šimon. Dále víme, že Eliška bydlí ve vile. Zjistíme tak dodatečné informace. Eliška totiž jistě nebude bydlet v bytě ani na chatě. A také víme, že ve vile nebude bydlet ani Pavla ani Šimon.

Tabulka 6:Demonstrace získání predikátu ¬má - vyloučením (Zdroj: vlastní)

		Osoba			Obydlí		
		Eliška	Pavla	Šimon	Chata	Vila	Byt
Osoba	Eliška	/	/	/	***-1***	1	***-1***
	Pavla	/	/	/	0	***-1***	0
	Šimon	/	/	/	0	***-1***	0
Obydlí	Chata				/	/	/
	Vila				/	/	/
	Byt				/	/	/

3. **Objekty jsou ve vztahu (predikát má) - tranzitivita**
    - Když platí *má (O_{11}, O_{12})* a zároveň *má (O_{12}, O_{13})*, pak platí *má (O_{11}, O_{13}).*
    - Příklad:
        - Vstup:
        - *má (O_{11}, O_{23}) „Eliška má myš.“*
        - *má (O_{23}, O_{12}) „Myš bydlí na chatě.“*
        - *Výstup:*
        - *má (O_{11}, O_{12}) „Eliška bydlí na chatě.“*

Tabulka 7: Demonstrace získání predikátu má – tranzitivita (Zdroj: vlastní)

		Osoba			Obydlí			Zvíře		
		Eliška	Pavla	Šimon	Chata	Vila	Byt	Pes	Myš	Ryba
Osoba	Eliška	/	/	/	***1***				1	
	Pavla	/	/	/						
	Šimon	/	/	/						
Obydlí	Chata				/	/	/		1	
	Vila				/	/	/			
	Byt				/	/	/			
Zvíře	Pes							/	/	/
	Myš							/	/	/
	Ryba							/	/	/

4. **Objekty nejsou ve vztahu (predikát ¬má) - tranzitivita**
    - Když platí *má (O11, O12)* a zároveň *relace_ne (O12, O13)*, pak platí *relace_ne (O11, O13).*
    - Příklad:
        - Vstup:
        - *má (O_{11}, O_{23}) „Eliška má myš.“*
        - *relace_ne (O_{23}, O_{12}) „Myš nebydlí na chatě.“*
        - *Výstup:*
        - *relace_ne (O_{11}, O_{12}) „Eliška nebydlí na chatě.“*

Tabulka 8: Demonstrace získání predikátu ¬má - tranzitivita (Zdroj: vlastní)

		Osoba			Obydlí			Zvíře		
		Eliška	Pavla	Šimon	Chata	Vila	Byt	Pes	Myš	Ryba
Osoba	Eliška	/	/	/	***-1***				1	
	Pavla	/	/	/						
	Šimon	/	/	/						
Obydlí	Chata				/	/	/		-1	
	Vila				/	/	/			
	Byt				/	/	/			
Zvíře	Pes							/	/	/
	Myš							/	/	/
	Ryba							/	/	/

Pomocí těchto čtyř metod můžeme, v případě úspěchu, zjistit, jestli jsou každé dva objekty spolu ve vztahu nebo jestli spolu nejsou ve vztahu. V případě neúspěchu nezjistíme o vztazích objektů nic. Použití těchto čtyř metod v algoritmu bude probíhat rekurzivně. V každé smyčce se nalezne několik nových platných predikátů a ty se přidají do množiny predikátů *P* a v další smyčce se díky nově přidaným predikátům můžou nalézt takové predikáty, které byly předtím nezjistitelně. Rekurze bude probíhat do té doby, než stavy množin *P* dvou po sobě jdoucích smyčkách jsou stejné a tedy se již nenalézají nové predikáty.

3.4. Postup řešení úlohy

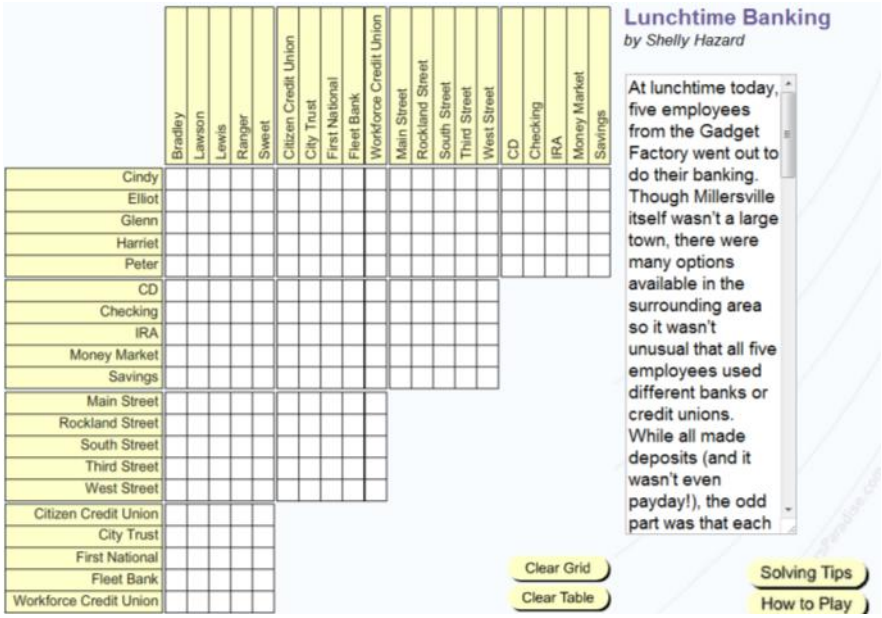
Když lidé řeší zebru, mohou použít čtverečkovaný papír, na nějž si přehledně mohou zobrazit všechny objekty a vztahy mezi nimi. Názvy objektů jsou vždy v záhlaví a kategorie můžeme oddělit tlustou čarou. Objekty po vzoru matice sousednosti budou v řádcích a sloupcích. Vztahy budou zobrazeny jako hodnoty v takto vytvořené matici. Jak jsme dříve zmínili, vztahy mezi objekty jsou vzájemné, tedy když by šlo o hrany

v matici sousednosti, byly by neorientované[6]. To způsobí, že matice sousednosti bude symetrická a tím pádem by při řešení hráči byla polovina matice na obtíž.

Některé internetové servery například dávají hráči k dispozici upravené matice, kde jsou jen ty pole, jen ty pole, která bude hráč potřebovat. Tento případ lze vidět na obrázku 8

Obrázek 8.

Obrázek 8: Herní pole s vynechanou částí matice (Zdroj: http://www.puzzlersparadise.com)



Hráč po přečtení každé indicie označí příslušné pole vztahu a to buď jako je ve vztahu nebo není ve vztahu. Je možné používat například tečku jako ano a křížek jako ne. Pole, o kterých nic nevíme, zůstávají prázdná. Po vyřešení hlavolamu bude každé pole zaplněno příslušnou značkou.

Pokud by úlohu řešil algoritmus, musíme postup formalizovat na pseudokód. Budeme používat metody z předchozí podkapitoly. Máme danou množinu predikátů P a množinu neznámých vztahů N.

[6] Například jeden vztah mezi objekty Jana a kočka, resp. Jana má kočku nám ve zmíněné matici označí vztah (tečkou, jedničkou apod.) na poli řádku Jana a sloupečku kočka a zároveň na poli řádku kočka a sloupečku Jana.

Pseudokód řešení úlohy (Zdroj: Peintner, 2011)

1. Vyber neznámý vztah *u*, pokud žádný není, běž na 5.
2. *jeSpojeny(u)* : ano / ne
    a. Ano: Ulož vazbu do *P* jako *má(u).*
    b. Pokračuj na 3.
3. *jeNespojeny(u)* : ano / ne
    a. Ano: Ulož vazbu do *P* jako ¬*má(u).*
    b. Ne: Pokračuj na 4.
4. Běž na 1.
5. Konec.

Obecný pseudokód řeší úlohu způsobem, že prochází postupně každý neznámý vztah mezi objekty a vždy ověří, jestli je mezi objekty platný predikát má, a po kladné odpovědi vždy ukládá informaci do množiny predikátů. Obdobně postupuje při zjištění, že objekty nejsou spojeny a tak platí predikát ¬má. Potom opět uloží informaci do množiny predikátů.

Jestliže je počáteční množina indicií P konzistentní, tj. zaručuje při správném postupu řešení nalezení všech vztahů, potom nám algoritmus s jistotou najde všechny správné vztahy. (Peinter, 2011)

V případě dvou predikátů (má a ¬má) je složitost algoritmu je v nejhorším případě $O(n^2)$, kde *n* je počet neznámých vztahů. (Peintner, 2011)

Pokud bychom používali predikátovou logiku a všechny skutečnosti upravili do formy logických výroků a vztahy mezi objekty bychom uvažovali jako predikáty

Např.: *„Petr má morče.“*

Tato věta se dá v predikátovém jazyce pojmout takto. *Petr* je subjekt a *má morče* je jeho predikát (atribut, vlastnost).

Jakmile bychom takto formulovali všechny skutečnosti, bylo by možné pro zjištění jednotlivých vztahů mezi objekty aplikovat rezoluční metodu.

3.5. Způsob vytvoření logické hry zebra

V této podkapitole jde o vytvoření minimální množiny predikátů *P*, která nám zajišťuje vyřešení úlohy.

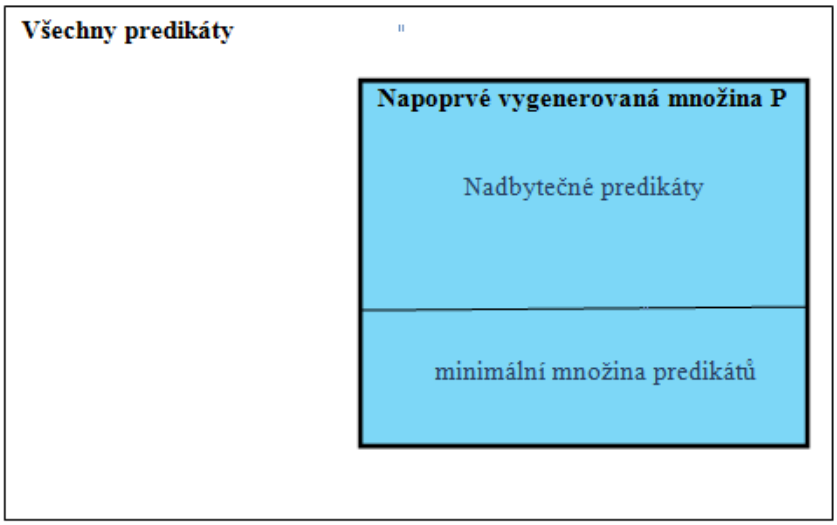
Jiný problém by byl nalezení nejmenší množiny indicií *P*, než by byla množina *P*, která by obsahovala nejméně prvků ze všech možných minimálních množin. Minimální množina s určitou vlastností je taková množina, pro kterou platí, že při odebrání jednoho prvku z této množiny se ztratí požadovaná vlastnost množiny a při přidání dalšího prvku do množiny vlastnost množiny zůstává. Z toho vyplývá, že nalezení nejmenší množiny *P* bude jistě mnohem složitější problém, než nalezení minimální množiny *P*. Bylo dokázáno, že problém nalezení nejmenší množiny *P* je NP-těžký. (Ryjáček, 2007)

Našim cílem tedy bude hledání množiny *P* minimální.

3.5.1. Nadbytečné predikáty

I když bychom určitým způsobem (například způsobem doplňování množiny indicií) nalezli množinu indicií, která nám zaručuje řešení, ještě nemáme jistotu, že se jedná o minimální množinu. Jestliže množina indicií je plněna náhodnými indiciemi a stále nezaručuje řešení až to chvíle, kdy po přidání určité indicie lze řešení nalézt, nemáme zaručeno, že vygenerovaná množina obsahuje pouze nutné prvky.

Obrázek 9: Vygenerovaná množina P a nadbytečné predikáty (Zdroj: vlastní)



Řekněme, že máme několik následujících indicií. Na počátku víme toto: „K obědu bude babička vařit polévku a tak bude připraveno jídlo“ a že „K obědu babička uvařila

knedlík a tak bude připraveno jídlo.“ A poslední informace je: „Jestliže bude připraveno jídlo, vnuk přijde navštívit babičku.“

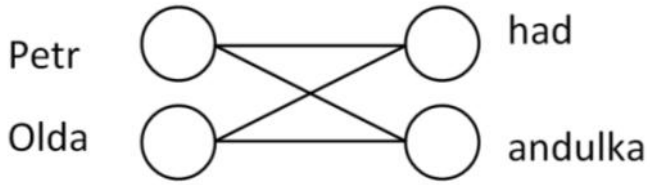
- K obědu bude babička vařit polévku. → Bude připraveno jídlo.
- K obědu babička uvaří knedlík. → Bude připraveno jídlo.
- Jestliže bude připraveno jídlo. → Vnuk přijde navštívit babičku.

Podívejme se na to, že konečný důsledek „Přijde navštívit babičku“ má několik příčin. Když přijdeme až k prvotním příčinám, tak jsou to: „K obědu bude babička vařit polévku.“ a „K obědu babička uvaří knedlík.“ Když bychom chtěli zjistit pouze, jestli přijde navštívit babičku, stačí nám k tomu jenom jedna ze dvou zmíněných příčin. Jedna z nich je tedy nadbytečná indicie.

Jestliže tedy nechceme generovat zbytečně jednoduché hlavolamy a chceme najít množinu *P*, která je minimální, musíme v ní prověřit všechny vztahy, které by mohli vyvozovat jiné vztahy, již v množině obsažené a tedy zbytečné.

3.5.2. Množina všech predikátů úlohy

Obrázek 10: Úloha s více řešeními (Zdroj: vlastní)



V dalších kapitolách budeme používat množinu všech indicií úlohy, tak tento problém nyní upřesníme. Opět budeme uvažovat pouze dva typy predikátů.

- predikát má → má (Petr, had) = „Petr má hada“
- predikát ¬má → ¬má (Petr, had) = „Petr nemá hada“

Nejdříve musíme zmínit, že pro jednu úlohu *U* existuje více možných řešení. Na uvedeném obrázku 10 je jednoduchá úloha v podobě bipartitiního grafu bez specifikace vztahů. Graf označuje, že Petr má hada nebo andulku a Olda má hada nebo andulku. Aby bylo možné úlohu řešit, musí mít graf konkrétní perfektní párování.

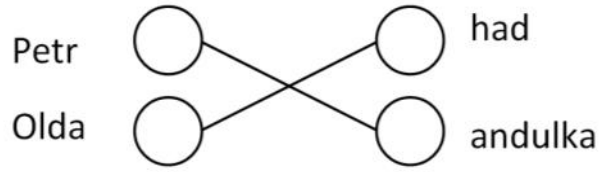
Úloha se dvěma kategoriemi a *n* objekty má $2 * n^2$ indicií. Vypíšeme seznam všech osmi možných indicií pro tuto úlohu.

1. Petr má hada.
2. Petr nemá hada.
3. Petr má andulku.
4. Petr nemá andulku
5. Olda má hada.
6. Olda nemá hada.
7. Olda má andulku
8. Olda nemá andulku.

Nyní by nastal problém, když bychom indicie generovali náhodně z celého tohoto seznamu. Níže je popsána modelová situace.

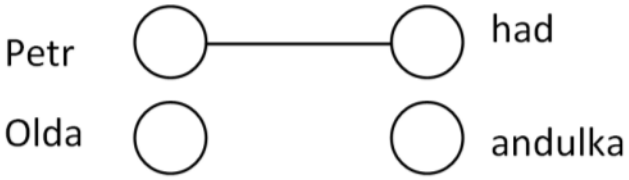
Nejdříve by mohla být náhodně zvolena 3. indicie, což by nechalo diagonální hranu (Petr, andulka) a automaticky by byly *vyloučením*, odebrány hrany (Petr, had) a (Olda, andulka).

Obrázek 11: Stav úlohy po aplikaci 3. indicie (Zdroj: vlastní)



Nyní bychom mohli považovat úlohu za vyřešenou. Pojďme se ale podívat, co by se stalo v případě, že bychom přidali další indicie, což by pro složitější úlohy bylo obvyklé. Správně by měli indicie jenom potvrzovat správné řešení. Aplikujeme nyní například indicii číslo 1 a dostáváme hranu (Petr, had) a vyloučením je automaticky odebrána hrana (Olda, had).

Obrázek 12: Úloha po aplikaci 3. a 1. indicie (Zdroj: vlastní)



V tomto okamžiku již graf úlohy nemůže mít perfektní párování, a tak se úloha stala neřešitelnou. Důvod, proč jsme dospěli k tomuto nechtěnému stavu, je fakt, že jsme indicie generovali z úplně všech možných indicií.

Zmíněná úloha má totiž dvě řešení. Ke každému z těchto řešení lze dojít pomocí disjunktní podmnožiny uvedeného seznamu indicií. Každé řešení má tedy svojí vlastní množinu indicií. Pro dvě kategorie po dvou objektech v tomto případě se indicie vzájemně vylučují (například první a druhá nebo první a pátá).

V dalších podkapitolách tedy budeme generovat množinu všech možných predikátů (indicií) vždy pro jedno řešení úlohy.

3.5.3. Způsob doplňování množiny predikátů

Na počátku máme *m* kategorií *n* objektů v každé kategorii a prázdnou množinu *P* a určitým způsobem (například zvolení náhodné kategorie a v ní náhodného objektu a k tomu druhý náhodně zvolený objekt z jiné kategorie) volíme některý ze všech možných vztahů jedné úlohy jeden vztah mezi objekty *v* a přiřadíme mu správný predikát, který povede k řešení dané úlohy – tedy buď predikát má(*v*) nebo ¬má(*v*).

Abychom mohli vybírat správné predikáty, potřebujeme mít, jak jsme v minulých kapitolách zmínili, množinu všech predikátů, kterou vygenerujeme z konkrétního řešení úlohy.

Po kroku přidání predikátu ověřujeme konzistentnost (množina je postačující k nalezení řešení) množiny *P* tím, že spustíme algoritmus k vyřešení úlohy[7]. Jestliže je řešení nalezeno, tak celkový algoritmus končí, a nalezli jsme množinu *P*, která nám určuje řešení.

Jestliže algoritmus nenašel řešení, pokračujeme opět přidáním dalšího vztahu jako v prvním odstavci.

[7] Zde je reprezentováno v tabulce algoritmu názvem algoritmu *řešitelné(U, P)*. Tento algoritmus je popsán v programové dokumentaci na přiloženém CD.

Algoritmus zvětšující se množiny *P*

1. Máme prázdnou množinu *P*.
2. Definujeme množinu všech možných indicií *X* pro úlohu *U*.
3. Odebereme z *X* náhodnou indicii a umístíme ji do *P*.
4. Spustíme *řešitelné(U,P) : ano / ne*
    a. Ano→ jdi na 5
    b. Ne → jdi na 4
5. Konec.

Po definování kategorií *K* a jejich objektů *O,* máme definovanou úlohu *U*. V kroku 2 definujeme množinu predikátů *P* jako prázdnou množinu. V dalším kroku pro dané řešení definujeme množinu *X* a uložíme do ní všechny možné predikáty mezi každými dvěma objekty *má*(o_i, o_j) a ¬*má*(o_i, o_j). V dalším kroku spustíme algoritmus pro vyřešení úlohy *řešitelné(U, P)* a dokud nebude řešitelná, budeme se vracet ke kroku 4 a přidávat do P další indicie. Jakmile je úloha řešitelná, algoritmus je u konce.

Nyní jsme se dostali do stavu, naznačeného v předchozí kapitole, kdy nejsme při generování náhodných indicií až do stavu, kdy je úloha řešitelná, schopni určit, jestli je minimální nebo nikoliv. Aby byla množina minimální, je skončení uvedeného algoritmu ještě nutné odstranit nadbytečné vztahy. Šlo by tedy o nový algoritmus, který by šel protisměru algoritmu zvětšujícího množinu *P*.

Na začátku tedy máme množinu *P*, pro kterou je úloha řešitelná. V množině se můžou vyskytovat některé indicie, které jsou nadbytečné. Z množiny *P* budeme postupně odebírat jeden prvek po druhém, přičemž při každém odebrání prvku budeme kontrolovat, jestli je úloha stále řešitelná., jestli není, vrátíme prvek zpět do množiny *P* a pokračujeme dalším prvkem, v opačném případě prvek odstraníme a pokračujeme následujícím prvkem. Postup je formalizován v následující tabulce.

Algoritmus odstranění nadbytečných predikátů

1. Platí *řešitelné(U, P) = ano*
2. Iterátor ukazuje na první prvek v *P*
3. Odebereme z *P* prvek označen iterátorem *R*.
4. Spustíme *řešitelné(U, P): Ano / Ne*
    a. Ano: Jdi na 3 (po odstranění *R* ukazuje na další prvek)
    b. Ne: Vrať prvek do *P* a inkrementuj *R*
5. Je R na konci?: Ano / Ne
    a. Ano: jdi na 6.
    b. Ne: Jdi na 3.
6. Konec (*R* prošel všechny prvky v *P*)

Tento algoritmus začíná úlohou *U*, která je řešitelná pomocí predikátů množiny *P*. Množinu *P* použijeme jako datovou strukturu s iterátorem, tedy například indexy. Iterátor byl nazván písmenem *R*. Počet indexů je konečný. Tolikrát algoritmus provede smyčku.

Odebereme tedy první prvek z množiny *P* a zkontrolujeme, jestli je úloha řešitelná[8]. Jestli je řešitelná, tak potom prvek nebudeme vracet a navrátíme se ke kroku číslo 3, kde odebíráme opět další prvek. V případě, že po odebrání prvku není úloha řešitelná, jedná se o nezbytný prvek a vrátíme ho do množiny *P*, inkrementujeme *R*, abychom se dostali k dalšímu prvku, a vracíme se ke kroku 3. Po každé inkrementaci proběhne kontrola, jestli již algoritmus nedošel na konec seznamu *P*.

Konec algoritmu je v okamžiku, kdy je v množině indicií *P* každý prvek otestován, jestli je nadbytečný a případně odstraněn, a množina *P* tak po proběhnutí algoritmu sestavena jen z nezbytných predikátů. Kdybychom odstranili jakoukoliv indicii, tak by množina *P* přestala uchovávat řešení a proto je tato množina *P* již minimální. Nemáme ovšem zaručeno, že takto získáme nejmenší množinu *P*. To bychom museli porovnat velikosti všech možných vygenerovaných minimálních množin, což by v případě průměrně veliké úlohy mohlo trvat velmi dlouho, viz začátek podkapitoly 3.5.

Tato metoda nám pro zadanou úlohu *U* zajišťuje vždy jiné zadání množiny indicií *P*. Je to tak díky prvnímu algoritmu, který vybírá predikáty z množiny *X* náhodně. V druhém algoritmu pro odstranění nadbytečných predikátů již vybíráme prvky postupně. Počet

[8] Algoritmus pro řešení úlohy je popsán v programové dokumentaci na přiloženém CD.

možných zadání úlohy závisí na množství kategorií a obsažených objektů. Počet kategorií a objektů s počtem různých zadání jedné úlohy *U* jsou ve vztahu přímé úměry. Čím více tedy bude kategorií a objektů, tím menší bude pravděpodobnost, že pro úlohu *U* vygenerujeme dvakrát za sebou stejné zadání.

3.5.4. Způsob ubírání z množiny predikátů

Druhý způsob získání zadání množiny indicií *P*, tedy vygenerování úlohy, je varianta, kdy množinu *P* naplníme již na začátku všemi predikáty, které jsou pro dané řešení možné. Jako v předešlém příkladě, budeme tuto množinu všech predikátů pro dané řešení značit *X*.

Chceme opět při každém spuštění algoritmu vygenerovat pro jednu úlohu *U* jiné zadání. Musíme tedy do algoritmu zahrnout prvek náhody.

Dále je uvedena tabulka, která se funkcemi podobá předcházejícímu algoritmu. Liší se ale počátečním stavem a reakcí na testování řešitelnosti.

Algoritmus zmenšující se množiny P

1. Máme úlohu *U*.
2. Definujeme množinu všech možných indicií *X*.
3. Zkopírujeme všechny indicie z X do množiny *P*.
4. Odebereme z *P* náhodnou indicii *i*.
5. Spustíme *řešitelné(U, P) : ano / ne*
    a. Ano: Jdi na 4.
    b. Ne: Jdi na 6.
6. Vrať indicii *i* do *P* a Konec.

Po definování úlohy *U* a množiny *P* a množiny *X* zkopírujeme všechny predikáty z množiny *X* do množiny *P*. Poté z množiny *P* odebíráme náhodné predikáty do té doby, dokud stále platí *řešitelné(U,P) = ano.* Až se po odebrání predikátu stane úloha neřešitelnou, tak vrátíme poslední predikát zpět do množiny *P* a algoritmus končí.

Tento postup nám při každém spuštění zaručí různé zadání množiny *P*. To je zajištěno náhodným odstraňováním predikátů z množiny *P*. Algoritmus nám ale nezaručuje, že je

množina *P* minimální, takže je nutné použít opět algoritmus pro odstranění nadbytečných predikátů jako v minulé podkapitole.

3.5.5. Srovnání algoritmů

Jestli první či druhý algoritmus vygeneruje složitější zadání je spíše věcí náhody a v průměru by obtížnost vygenerovaných zadání měla být obdobná. Algoritmy by se však mohli lišit v délce výpočtu. Zatímco metoda, která do množiny *P* přidává predikáty, pro jednoduchou úlohu 2 kategorie po 2 objektech odhalí jeden další predikát a úloha se stane řešitelnou, tak druhá metoda má na počátku 4 indicie a musí odebrat všechny 4, než se stane množina *P* nedostatečnou a algoritmus skončí.

Pro lepší představivost daný problém ilustrujeme v následující tabulce.

Tabulka 9: Jednoduchá zebra (Zdroj: vlastní)

	Petr	Jana	Pes	Kočka
Petr	/	/	1	-1
Jana	/	/	-1	1
Pes	1	-1	/	/
Kočka	-1	1	/	/

Řešením této úlohy je zjištění, že Petr vlastní psa a Jana vlastní kočku. V této úloze postačuje v případě prvního algoritmu vybrat jediný predikát. Například to může být jeden z těchto:

- Petr nemá kočku.
- Jana nemá psa.

Pro druhou a poslední variantu zadání by algoritmus vybral z negací předchozích výroků.

V případě druhého algoritmu by na počátku byly do množiny *P* vloženy následující predikáty:

1. Petr má psa
2. Petr nemá kočku.
3. Jana má kočku.
4. Jana nemá psa.

Z toho plyne, že první algoritmus přidávající predikáty do množiny *P* nalezne pro tuto jednoduchou úlohu řešení rychleji než algoritmus, který predikáty ubírá. Proto bude v implementaci použit algoritmus, který predikáty přidává.

5. Návrh programu

5.1. Požadavky na funkčnost programu

Hlavním úkolem aplikace je generování zadání pro úlohy typu zebra definované v předcházející části této práce.

Program by měl umět na základě definování několika kategorií a jejich objektů sestavit řešení úlohy a k tomuto řešení posléze náhodně generovat různá zadání, které povedou k danému výsledku. Zadání se skládá z několika druhů indicií. Počet indicií každého druhu je přímo úměrný počtu kategorií a počtu objektů v kategoriích.

Tyto vygenerovaná zadání by měly být přehledně odděleny a jako celky by měli být uživateli k dispozici, aby si je mohl zkopírovat a dále s nimi pracovat (poslat E-mailem, vytisknout, uložit na disk).

Program by měl být spustitelný na běžných počítačích a paměťové a výkonnostní nároky na generování běžných úloh by neměly přesahovat výkon běžných počítačů. Na těchto počítačích by běh algoritmu měl správně pracovat tak, aby to trvalo snesitelnou dobu. Výpočetní čas by měl být v řádu sekund, maximálně desítek sekund. Program byl odlaďován a zkoušen a počítači s běžným (na dnešní dobu nižším) výkonem, který je uveden v této tabulce.

Tabulka 10: Výkonnostní parametry zkušebního PC (Zdroj: vlastní)

Procesor	Intel Pentium 4, CPU 2800 MHz
Paměť RAM	1500 MB
Operační systém	Windows 7, 32bit

Dále by program měl přehledně uživateli zobrazovat všechny potřebné komponenty, nutné pro běh programu, komunikovat s uživatelem a vytvořit prostředí pro plné využití možností programu.

5.2. Volba programovacích prostředků

5.2.1. Volba programovacího jazyka

Pro implementaci aplikace byl použit programovací jazyk Java. A to pro jeho několik kladných vlastností, mezi něž patří přenositelnost na jiné platformy, moderní programový interface a popularita jazyka. Jazyk Java je dnes totiž hojně používán v podnicích i pro výuku programování na středních a vysokých školách. Program tedy díky zmíněné vlastnosti přenositelnosti může být spuštěn jak na Windows, tak na Unix a dalších operačních systémech. Pro spouštění programů vytvořených v Javě je třeba mít pouze na daném stroji nainstalovaný virtuální stroj Javy (Java Virtual Machine). Výhoda Javy je také v tom, že dovoluje používat objekty, což usnadňuje a zpřehledňuje práci programátora. V tomto programovacím jazyce je také umožněno vytvořit vhodné uživatelské rozhraní pro vytvoření uživatelsky komfortního prostředí, které umožní uživateli efektivně používat napsané programy.

Objektový programovací jazyk Java byl vyvinut na základě jazyka C++ a vytvořila jej firma Sun Microsystems. Java byla veřejně uvedena roku 1995. Javu tvoří již zmíněný Java Virtual Machine, který zajišťuje vazbu na hardware a zároveň kompiluje napsaný kód. Druhý základní kámen Javy je Java Core API, což je aplikační programové rozhraní, které nám umožňuje používat přes tisíc základních knihovních tříd, což jsou předem vytvořené a z hlediska programové efektivity optimalizované kusy kódu, které je možno v našich programech používat. (Herout, 2000)

Javu je možné používat v Java SE (Standart Edition), kterou používám v této práci, Java ME (Mobile Edition), pro programování software na mobilní zařízení a Java EE (Enterprise Edition), která se používá pro vývoj rozsáhlých firemních aplikací.

5.2.2. Programovací nástroje

Pro vytvoření programu v Javě je třeba napsat kód v náležité formě jazyka, soubor uložit s koncovkou „.java“, zkompilovat (v příkazové řádce příkazem *javac*) a spustit (v příkazové řádce příkazem *java).* K tomu nám ve Windows postačí jen poznámkový blok (*notepad*) a přírazový řádek (*command line*).

Ale vytvářet běžné programy by bylo s takto primitivními nástroji značně ztíženo. Kód by byl nepřehledným množstvím textu, ve kterém se lze jen těžko zorientovat.

Pro větší využití programátorova potenciálu se dnes používají tzv. IDE, což je zkratka v angličtině: Integrated Developer Environment, tedy integrované vývojářské prostředí.

Příkladem některých vývojářských prostředí Javy jsou tyto systémy (Zdroj: programmerworld.net):

- Eclipse - zdarma
- NetBeans - zdarma
- IntelliJ – $ 499
- JCreator – zdarma
- JBuilder – $ 499
- JEdit – zdarma

Autor byl z vlastní zkušenosti obeznámen se dvěma systémy. Eclipse a NetBeans. Systém NetBeans od autorů jazyka Javy – firmy Sun Microsystems – sice oplývá výhodami jako přehlednost, mnoho možností automatického generování kódu a především nástroje pro automatické vytváření grafického uživatelského prostředí, ale autor se přiklání k systému Eclipse, který je dnes velmi používaný, je přizpůsobitelný a i když mu chybí nástroj pro vytváření grafického uživatelského rozhraní, tak se také díky svému prvotřídnímu designu stal nástrojem číslo jedna pro vývoj aplikace této práce. Kód napsaný v IDE Eclipse je dle autorova názoru lépe ovladatelný než kód z NetBeans, jelikož se nedělí na napsanou (editovatelnou) část a generovanou (needitovatelnou) část.

Veškerý kód napsaný v Eclipse má zvýrazněné syntakticky důležitá slova, zvýrazňuje aktuálně vybraná klíčová slova, je možný tzv. refactoring, kdy je například změna v přejmenování provedena na všech úrovních programu, kde se slovo vyskytuje. Vývojové prostředí také umožňuje některé jednotvárné kusy kódu generovat a hlavně je zde k dispozici ladící program umožňující krokovat jednotlivé části programu, tzv. debugger.

5.3. Návrh aplikace

V následující části práce byl pojem predikát nahrazen slovem indicie. Toto slovo zastupuje pojem predikát používaný v předcházející části práce. Indicie je zde tedy predikát, který vždy informuje o určité vlastnosti jednoho objektu zahrnující jiný objekt.

5.3.1. Definování kategorií a objektů

Program byl jednoduše navrhnut tak, aby umožnil uživateli zadat počet kategorií a počet objektů v každé kategorii. Posléze uživatel pojmenuje všechny objekty. Například když uživatel zadá tři kategorie a tři osoby, tak vymyslí jména pro tři osoby, obydlí a zvířata těchto tří osob.

Uživatel pouze vybere počet kategorií. Kategorie jsou potom přiřazeny postupně od začátku předem definovaného seznamu.

Tabulka 11: Seznam použitých kategorií v aplikaci (Zdroj: vlastní)

1	2	3	4	5	6	7
osoba	*obydlí*	*zvíře*	*jídlo*	*jazyk*	*pití*	*oblečení*

Jestliže tedy uživatel zadá na začátku tři kategorie, aplikace potom po něm bude vyžadovat pojmenování jednotlivých objektů osob, obydlí a zvířat a dále bude pracovat s těmito třemi kategoriemi.

Toto řešení bylo zvoleno vzhledem k úspoře času uživatele a kompaktnosti programu. Není tak nutné vypisovat další množství informací nutných pro běh programu. Aplikace zvolí kategorie automaticky a uživatel tak může rovnou zapisovat jména jednotlivých objektů, ke kterým je již kategorie přiřazena.

Z pohledu programátora je třeba vytvořit pro uživatele několik komponent, aby bylo možno zadat počet kategorií a počet objektů.

Na konci tohoto kroku by mohl uživatel mít definovány například tyto hodnoty:

- počet kategorií = 3
- počet objektů kategorie = 3

Tabulka 12: Příklad definování názvů objektů (Zdroj: vlastní)

Osoba:	Michala	Eva	Zuzka
Obydlí:	Bílý dům	Modrý dům	Vila
Zvíře:	Bernardýn	Kočka	Opice

Mezi takto definovanými objekty nyní necháme uživatele vytvořit vazby, jejichž odhalení bude předmětem vyřešení hlavolamu.

Tabulka 13: Uživatel definuje vztahy mezi objekty (Zdroj: vlastní)

Osoba:	Michala	Eva	Zuzka
Obydlí:	Bílý dům	Modrý dům	Vila
Zvíře:	Bernardýn	Kočka	Opice

Až uživatel takto definuje vztahy, tak tím stanoví, jak bude vypadat vyřešená úloha. Každá osoba vlastní právě jedno zvíře a bydlí v právě jednom domě. Každý musí mít přiřazené jiné obydlí a jiné zvíře.

Nyní máme zadané řešení zadané úlohy. Pro takovou úlohu se budeme snažit vygenerovat zadání. V následující podkapitole se budeme detailněji zabývat zadáním a autorem navrženým algoritmem pro jeho generování.

5.3.2. Generování náhodného zadání

Zadání úlohy se skládá z určité – různě veliké množiny indicií. V předchozí části jsme definovali několik druhů indicií. Byly to indicie dvou typů (má, ¬má):

1. **má**: „Eliška má byt“
2. **¬má**: „Eliška má chatu“

Pro větší rozmanitost úlohy autor přidal ještě další tři typy predikátů. Ty byly definovány takto.

3. **má_vedle**: „Eliška má obydlí vedle chaty.“

4. **má_napravo**: „Eliška má obydlí napravo od chaty.“
5. **má_nalevo**: „Eliška má obydlí nalevo od vily.“

Každý predikát je upraven v závislosti na kategorii objektu, kterého se týká. Funkčnost je ale v případě jakékoliv kategorie stejná, tak budou predikáty rozděleny pouze na těchto 5 typů, podle nichž budou implementovány příslušné metody.

Objekty tedy musí být zřetelně definovány tak, aby uživatel viděl, kde se nalézají jednotlivé objekty. Systém hry je navržen tak, že jsou objekty seřazeny a zobrazeny hráči spolu se zadanou množinou indicií, což je tedy součástí zadání.

Můžeme mít například úlohu se čtyřmi objekty v každé kategorii, kde kategorie jsou osoba a obydlí. Program nám po zadání názvů objektů v první části zobrazí, jak jdou objekty po sobě například takto:

Objekty kategorie OSOBA:

1. Jindřich
2. Jan
3. Jiřina
4. Jitka

Objekty kategorie OBYDLÍ:

1. Červený dům
2. Modrý dům
3. Bílý dům
4. Žlutý dům

Při tomto pořadí objektů nyní hráč bude moci zohledňovat indicie zadání. Mějme tedy například zadány tyto indicie:

indicie:

1. Jiřina má obydlí napravo od Modrého domu.
2. Jindřich má obydlí vedle Modrého domu.

První indicie nám odhalí, že Jiřina nebydlí v Modrém ani v Červeném domě ale napravo od něj, tedy v Bílém nebo Žlutém domě.

Druhá indicie nám řekne, že Jindřich bydlí bílém nebo červeném domě. Ostatní domy můžeme vyloučit.

Dále bude vysvětlen navržený algoritmus pro generování minimální množiny zadání postačující k vyřešení dané úlohy.

1. fáze: generování všech indicií

V první fázi algoritmus k danému řešení úlohy (všechny definované výsledné vztahy) vygeneruje všechny možné indicie, které nám poskytují všechny námi definované informace (množinu indicií) o každém objektu. Zde pro příklad uvedeme všechny indicie o objektu kategorie osoba, Jindřich.

1. Jindřich má červený dům.
2. Jindřich nemá modrý dům.
3. Jindřich nemá bílý dům.
4. Jindřich nemá žlutý dům.
5. Jindřich má obydlí nalevo od modrého domu.
6. Jindřich má obydlí nalevo od bílého domu.
7. Jindřich má obydlí nalevo od žlutého domu.
8. Jindřich má obydlí vedle modrého domu.

Algoritmus díky znalosti řešení dopředu ví, že Jindřich bydlí v červeném domě a zná také všechny ostatní vztahy. Tyto indicie algoritmus vygeneruje pro každý vztah mezi objekty. Pro tento příklad pro dvě kategorie a čtyři objekty algoritmus vygeneruje 34 indicií.

2. fáze: zredukování množství indicií

V této fázi používáme na první pohled jednoduchý algoritmus, který ale vyžaduje jiný algoritmus – řešitele úlohy (dále budeme používat pouze slovo řešitel). Řešitel je algoritmus, který je schopen při zadané množině indicií úlohy odpovědět, zda je úloha řešitelná či nikoliv.

Zjednodušená posloupnost příkazů algoritmu redukujícího množinu všech indicií je v následujícím schématu:

Algoritmus redukce množiny indicií

1. *I_vše* = všechny indicie k řešení úlohy
2. *I_redukovaná* = *I_vše*
3. Zamíchej prvky *I_redukovaná*
4. Odeber indicii z *I_redukovaná*.
5. Řešitelná (*I_redukovaná*): ano / ne
    a. ano → jdi na 4.
    b. ne → jdi na 6.
6. Vrať poslední odebranou indicii do *I_redukovaná*.
7. Konec.

Nejdříve algoritmus uloží do proměnné *I_redukovaná* všechny indicie. Potom tuto množinu zamícháme a postupně odebíráme indicie a pokaždé testujeme, zda je úloha ještě řešitelná. Ve chvíli, kdy už úloha není řešitelná, přejdeme do dalšího kroku a vrátíme poslední odebranou indicii zpět do proměnné *I_redukovaná*. Potom máme redukovanou množinu indicií k danému řešení úlohy.

Tato množina stále není minimální, jelikož jsme mohli během odebírání indicií přicházet o důležité indicie a zbytečné v proměnné mohly stále zůstávat. V množině se tedy stále můžou nacházet nadbytečné indicie, které je třeba odstranit v dalším kroku.

3. fáze: minimalizace množství indicií

V tomto kroku použijeme obdobný algoritmus jako v předešlém kroku. Je o málo složitější. Předchozí algoritmus odstranil většinu zbytečných indicií rychlejším způsobem a následující algoritmus bude prohledávat zbývající prvky množiny podrobnějším postupem. Obrazně řečeno, předchozí algoritmus fungoval jako hrubé síto odstraňující nadbytečné indicie a následující algoritmus funguje jako jemné síto, jehož výstupem je minimální množina indicií, potřebná pro vyřešení úlohy. Posloupnost příkazů je v následujícím schématu.

Algoritmus minimalizace množství indicií

1. *I_minimální = I_redukovaná; Iterátor = 0;*
2. Odeber indicii z *I_minimální*, na kterou ukazuje *iterátor*.
3. Řešitelná (*I_minimální*): ano / ne
    a. ano → jdi na 2
    b. ne → vrať indicii, inkrementuj *iterátor*, a jdi na 2.
4. *Iterátor* došel na konec seznamu *I_minimální*
5. Konec.

Zde algoritmus prochází celou množinu zbývajících indicií prvek po prvku a pokaždé testuje, zda je úloha řešitelná s aktuální množinou indicií. Jestliže je řešitelná, prvek ponecháme odstraněn a všechny prvky seznamu se posouvají dopředu na místo odstraněného prvku a *iterátor* ukazuje na následující prvek, který opět odstraníme a testujeme řešitelnost. Jestliže je úloha neřešitelná, tak prvek vrátíme a iterátor inkrementujeme, který pak ukazuje na další prvek. Takto prověříme celý seznam a dostaneme minimální množinu indicií potřebnou pro vyřešení úlohy.

5.4. Programová struktura aplikace

Kód v Javě je členěn na třídy. Jsou to kusy kódu s určitou nadřazenou vlastností pro všechny procedury a atributy. Na obrázku 13 můžete vidět schéma tříd programu – tzv. UML diagram. Třídy jsou zobrazeny jako svislé obdélníky s tučným názvem nahoře. Pod názvem třídy jsou atributy a metody.

- atribut – Jde o samostatný datový typ nebo objekt popisující určitou část třídy. Atributy třídy Auto by mohly být například průměrná rychlost, dojezd, palivo, poznávací značka, majitel. U složitějších tříd atributy slouží k uchování informací, které jsou důležité pro operace třídy. Atributy mohou být mimo kód vlastní třídy získány a nastaveny. Atributy začínají malým písmenem.
- metoda – Jedná se o samostatný blok kódu, který může mít jeden, žádný nebo více parametrů, které ovlivní výstupní hodnotu, kterou může být objekt nebo některý datový typ. Metody mohou být také datového typu void, kdy po dokončení své činnosti nevrací žádnou hodnotu. Metody začínají malým písmenem a jsou vždy zakončeny parametrickými závorkami. V případě, že jsou

bezparametrické, je za názvem pouze otevřená a zavřená závorka. V případě konstruktoru je název metody shodný s názvem třídy a tak je první písmeno velké.

V tomto programu pro generování různých zadání pro úlohy typu zebra definované předchozí části práce byl kód rozdělen intuitivně podle oken grafického uživatelského rozhraní a podle funkčnosti částí aplikace. Základní kód programu je členěn do šesti tříd. Ostatní třídy jsou pouze pomocného charakteru.

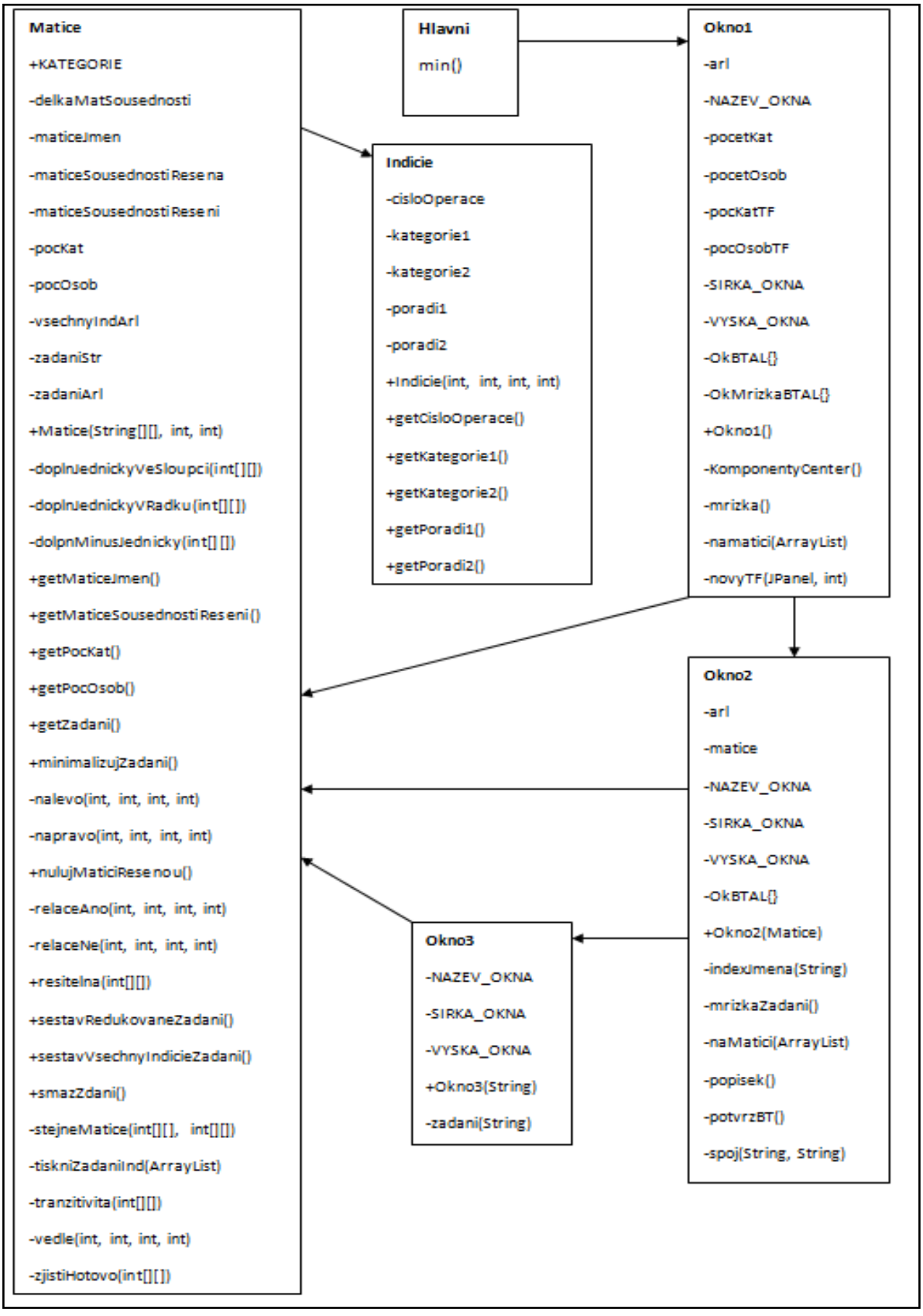
Aplikace se spouští metodou main umístěné ve třídě Hlavní. Metoda main spouští konstruktor třídy Okno1.

Program tedy nejprve vytvoří první okno (třída Okno1) pro zadání počtu kategorií, počtu objektů v kategorii a pojmenování jednotlivých objektů. Potvrzením program vytvoří jiné okno (třída Okno2) pro rozvržení výsledných vazeb mezi objekty zadanými v předchozím okně. Potvrzením vazeb se otevře třetí okno (třída Okno3), která vypíše vygenerované zadání.

Všechny tyto třídy ukládají vstupy od uživatele a nechávají provádět výpočty za pomoci třídy, která se jmenuje Matice. Jméno bylo zvoleno podle základní vytvořené datové struktury matice, která udržuje všechny informace o zadané úloze.

V následujících podkapitolách bude rozebrána funkcionalita jednotlivých tříd. Kromě toho budou popsány výhody některých použitých předdefinovaných knihovních tříd Java API (aplikační programové rozhraní).

Obrázek 13: Programová struktura aplikace – UML diagram (Zdroj: vlastní)



5.4.1. Třída Okno1

Tato třída hned při prvotním spuštění programu vytvoří malé okno. K tomu je použita přeprogramovaná třída z programové rodiny Swing. Swing je rozsáhlý soubor tříd sloužících pro vytváření grafického uživatelského rozhraní. Pro třídu Okno1 byly použity tyto komponenty.

- **JFrame** – Toto je základní komponenta pro vytvoření každého okna ve Swingu. Je potřeba nastavit výšku a šířku v pixelech, řídící režim rozložení komponent v okně (Layout Manager), který každou vloženou komponentu umístí a přizpůsobí podle svých pravidel. Dále se nastaví název titulku v liště záhlaví okna a metodou setVisible se nastaví viditelnost okna.
- **JPanel** – Komponenta, která je pouze ohraničenou oblastí pro vkládání dalších komponent je JPanel. Jeden JFrame, tedy okno, může mít v sobě několik komponent typu JPanel. Tyto panely jsou rozmístěny a přizpůsobeny podle stanovaného Layout Manageru a obsahují další komponenty, které už mají většinou nějakou viditelnou vlastnost (tlačítko, textové pole, informativní text či menu).
- **JLabel** – Komponenta JLabel je jednoduchý textový popisek. Parametrem při vytváření popisku je textový řetězec. Dále je možné nastavit font. Pro formátování textu je možné vstupní textový řetězec upravit jako html kód. Funkcionalita popisku je informovat uživatele o průběhu procedur v aplikaci, o výsledcích výpočtů, popsání jiných komponent apod.
- **JTextField** – Textové pole slouží zpravidla pro načtení textu, který zadal uživatel. Komponenta může mít nastavenou počáteční textovou hodnotu a může mít zakázané úpravy, což je graficky poznat zšednutím místa pro editaci.
- **JButton** – Tlačítko slouží pro potvrzení vstupů, odstartování určité metody v programu nebo také jako ovládaví prvek programu. Každé tlačítko by mělo být pojmenováno nebo by mělo obsahovat ikonu, aby byla zřejmá jeho funkčnost. JButton je typickou třídou, která téměř vždy nastavuje posluchač. Nejčastějším posluchačem tlačítka je ActionListener, který stanoví, co se bude dít po stisknutí tlačítka.

Třída Okno1 vytvoří klasické okno s delší vertikální stranou, se dvěma textovými poli, kde je každé popsáno popiskem a pro potvrzení zadaných hodnot v těchto polích je pod nimi tlačítko. Uživatel do těchto dvou textových polí zadá celé a kladné číselné hodnoty pro nastavení atributů třídy Okno1 pocKat a pocOsob. Atribut pocKat je počet kategorií a pocOsob znamená počet objektů v každé kategorii. V hlavolamu jsou objekty první kategorie vždy osoby, tak je atribut pro lepší vizuální představivost pojmenován pocOsob. Do textových polí nelze zadat nic jiného než číslice. Není tedy možné zadat ani pomlčku ani čárku. Podmínka zadání přirozeného čísla je tedy splněna. Po stisknutí tlačítka OK se nastaví zmíněné atributy a vykreslí se požadovaný počet textových polí pro zadání názvů objektů.

Jestliže uživatel zadal tři kategorie po čtyřech objektech, tak bude třeba vytvořit dvanáct textových polí pro zadání názvů objektů. Jelikož se zde pracuje s proměnlivým počtem netriviálních objektů, je vhodné použít důmyslnější datovou strukturu než pole. K tomu posloužila třída ArrayList.

- **ArrayList** – Jedná se o seznam specifikovaných objektů, kde má každý vložený prvek své očíslování. Prvky lze na libovolném místě vložit nebo odstranit. Při těchto změnách jsou ostatní prvky vzápětí automaticky přečíslovány. ArrayList čísluje tradičně od nuly (0, 1, 2, 3 …).

Do tohoto seznamu (pojmenovaném arl) vložíme požadovaný počet komponent TextField a pomocí dvou for-cyklů (pro kategorie jako řádky a jednotlivé objekty jako sloupečky) vložíme textová pole do panelu s řízeným rozložením komponent do mřížky. Pro zviditelnění těchto nových komponenty nastavíme viditelnost okna pomocí metody třídy JFrame setVisible na hodnotu false a hned true. To způsobí opětovné vykreslení okna a jeho všech komponent.

Až uživatel zadá názvy pro objekty všech kategorií, může stisknout přidružené potvrzovací tlačítko. Potom bude vytvořen nový objekt třídy Matice a s ním také nová instance třídy Okno2.

5.4.2. Třída Okno2

Potom, co se objeví druhé okno, se první okno zavře. Druhé okno je tvarem podobné prvnímu oknu. Celá tato třída od okamžiku jejího vytvoření pracuje s vytvořenou instancí třídy Matice, kterou dále využívá a doplňuje jí potřebnými vstupy získanými od uživatele.

V horní části okna jsou uvedeny instrukce pro uživatele. Program od uživatele požaduje zadání vztahů mezi jednotlivými objekty. Objekty jsou posléze vypsány v pořadí, v jakém jsou za sebou.

Způsob zadání vazeb mezi objekty je zvolen jako matice rozbalovacích seznamů.

- JComboBox – Rozbalovací seznam je určen pro provedení výběru mezi několika možnostmi popsanými textem. Vzhled je podobný textovému poli, kde je navíc na pravé straně šipka dolů. Při klepnutí na šipku se otevře seznam možností, na které je možné kliknout. Při provedení výběru se seznam opět uzavře.

V matici těchto rozbalovacích seznamů jsou řádky definovanými kategoriemi a sloupce představují jednotlivé kategorie. Pro každý sloupec tedy uživatel může zadat přidružené objekty z ostatních kategorií. Ve chvíli, kdy jsou v každém rozevíracím seznamu vybrány jiné názvy objektů, může uživatel stisknout tlačítko OK pro potvrzení jeho vstupů. Pro realizaci zobrazení proměnlivého počtu rozbalovacích seznamů byl použit obdobné postup jako v minulém případě zobrazování proměnlivého počtu textových polí. Opět byla použita efektivní datová struktura seznamu ArrayList.

Po stisku tlačítka se odehraje množství operací. Spustí se většina kódu ve třídě matice využívající třídu indicie a po všech výpočtech se zobrazí výstupy ve třídě Okno3. Okno 2 zůstane otevřené pro opakované generování náhodného zadání k zadané úloze. Po zavření druhého okna bude program ukončen.

5.4.3. Třída Okno3

Třída třetího okna je nejjednodušším oknem ze všech ostatních. Při vytváření okna musí být uveden textový parametr, který obsahuje výsledné indicie vygenerovaného zadání. Tento dlouhý řetězec je vložen do textové plochy.

- JTextArea – Tato komponenta je určena pro delší texty, které se můžou měnit programově nebo uživatelem. Text lze odtud také kopírovat tradičním způsobem, tedy stiskem kombinace kláves Ctrl + C.

Textová plocha j je komponenta vyplňující veškerý prostor třetího okna. Text uvnitř může být zkopírován do schránky a dále použit v textovém editoru a být vytisknut.

5.4.4. Třída Indicie

Tato třída je jedna z nejjednodušších. Obsahuje pouze informace o indicii. Konstruktor třídy obsahuje pět celočíselných parametrů. Každá indicie potřebuje uvést informaci ohledně dvou objektů. Každý objekt se dá popsat jako objekt v určitém pořadí v dané kategorii. To jsou dvakrát dva číselné údaje. K tomu je ještě přidána informace o daném vztahu mezi objekty, který je předdefinovaný také jako číslo od jedné do pěti podle typu operace, kterou dva objekty provedou při jejich zohlednění při řešení úlohy.

Každá indicie je tedy v programu chápána takto.

- Pořadí prvního objektu
- Pořadí druhého objektu
- Kategorie prvního objektu
- Kategorie druhého objektu
- Číslo operace provedené objekty

Dále třída obsahuje jenom metody pro získání hodnot pěti zmíněných celočíselných atributů, tedy tzv. getry.

5.4.5. Třída Matice

Třída matice obsahuje množství atributů a metod pro zadání úlohy, její řešení a generování zadání s minimální velikostí.

Začněme popsáním základních atributů třídy.

- **maticeSousednostiReseni** – Matice sousednosti řešení představuje řešení zadané úlohy uspořádané do matice. Matice sousednosti lze interpretovat jako matici sousednosti z teorie grafů. Tento atribut je datového typu matice celých čísel. V Javě je tento datový typ označen int[][]. Matice je čtvercová s počtem sloupců, který je roven součinu počtu kategorií s počtem objektů v kategorii. Jednotlivé hodnoty polí matice jsou rovny buď 0 (neznáme relaci) nebo 1 (relace mezi objektem v řádku a sloupečku existuje). Hodnota atributu je nastavena ve chvíli, kdy uživatel potvrdí stiskem tlačítka OK ve druhém okně svůj výběr relací mezi objekty kategorií, neboli v okamžiku, kdy dodefinuje řešení.
- **maticeSousednostiResena** – Co se týče datového typu, je opět o celočíselnou čtvercovou matici rozměru [počet kategorií * počet objektů v kategorii] x [počet kategorií * počet objektů v kategorii]. Jedná se o matici sousednosti řešené úlohy, tak jsou hodnoty polí kromě 0 (neznáme relaci) a 1 (relace je) rozšířeny o -1, což znamená zjištění, že relace mezi objektem v řádku a sloupci není. Tato matice se vyplňuje algoritmem řešícím úlohu pomocí zadaných indicií.
- **maticeJmen** – Jedná se o matici řetězců (String) rozměrů [počet kategorií] x [počet objektů v kategorii]. Hodnoty polí matice jmen jsou nastaveny při konečném potvrzení prvního okna potom, co uživatel zadá počet kategorií, počet objektů v kategorii a následovně vyplní jména všech objektů všech kategorií. Matice jmen se používá k zapamatování a rozpoznání všech jmen objektů. Použitá je na konci programu, kdy jsou indicie vypsány ve formátu čitelných vět. Do té doby pracuje program pouze s čísly.
- **vsechnyindicieArl** a **zadaniArl** – Jedná se o dva seznamy proměnlivé délky, kde do prvního seznamu jsou vloženy všechny nalezené indicie (typ objektu indicie) k dané úloze a druhý seznam je prázdný a postupně jsou do něj

vkládány indicie ze zamíchaného prvního seznamu, dokud se úloha nestane řešitelnou na základě indicií v seznamu zadaniArl.

Hlavní náplní třídy matice je generátor zadání pro danou úlohu, kterou zadává uživatel ve vytvořeném uživatelském rozhraní (třídy Okno1, Okno2). Jsou nastaveny matice sousednosti řešení a jména objektů. Matice sousednosti – řešená zatím zůstává vynulovaná.

V prvním kroku jsou do seznamu všechnyindicieArl vloženy všechny možné indicie. Program prohledává všechny pole matice sousednosti řešení a postupně podle nich doplňuje všech pět typů indicií. Tyto procesy řídí metoda sestavVsechnyindicieZadani().

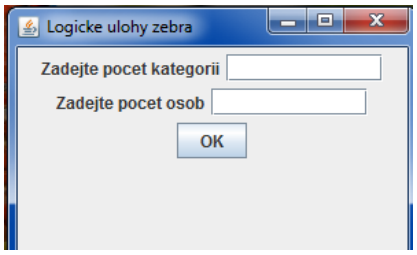
Ve druhém kroku se zamíchá seznam všechnyindicieArl a postupně jsou z něj kopírovány indicie do seznamu zadaniArl do té doby, dokud je úloha neřešitelná na základě indicií v seznamu zadaniArl. Program v tuto chvíli je řízen metodou sestavRedukovaneZadani(). Pro zjištění, jestli je úloha řešitelná je spouštěna metoda resitelna(int[][] matice).

Ve třetím kroku je spuštěna metoda minimalizujZadani(), která probírá celý seznam zadaniArl prvek po prvku. Každou indicii odstraní a vrátí ji v případě, že by se úloha stala neřešitelnou. Takto ověří všechny prvky a odstraní všechny prvky, které jsou zbytečné. Zbude minimální zadání, které bylo hledáno.

6. Uživatelská příručka

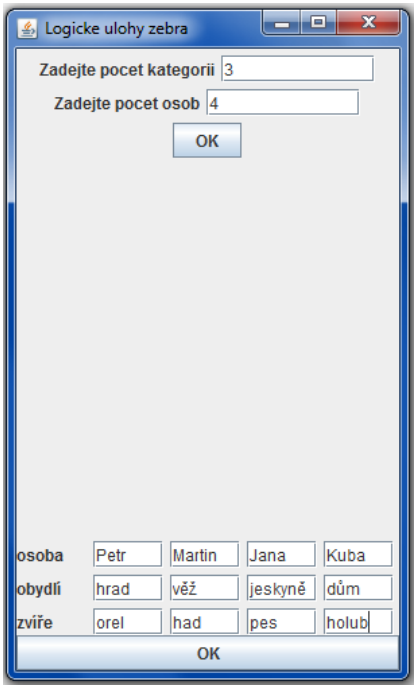
V prvním kroku uživatel zadá počet kategorií a počet osob, kterých se bude úloha týkat.

Obrázek 14: Zadání počtu kategorií a osob (Zdroj: vlastní)



Jakmile uživatel potvrdí hodnoty stiskem tlačítka OK, plocha okna se překreslí a ve spodní části se objeví daný počet testových polí. Jestliže jsme zadali tři kategorie, budou kategorie osoba, obydlí a zvíře. Zadané čtyři osoby potom do každé kategorie zařadí čtyři objekty. Celkem je tedy k vyplnění dvanáct textových polí. Pojmenování objektů závisí na představivosti uživatele.

Obrázek 15: Pojmenování objektů (Zdroj: vlastní)



Jakmile uživatel dokončí pojmenování objektů, stiskne spodní potvrzovací tlačítko, čímž přejde do dalšího okna.

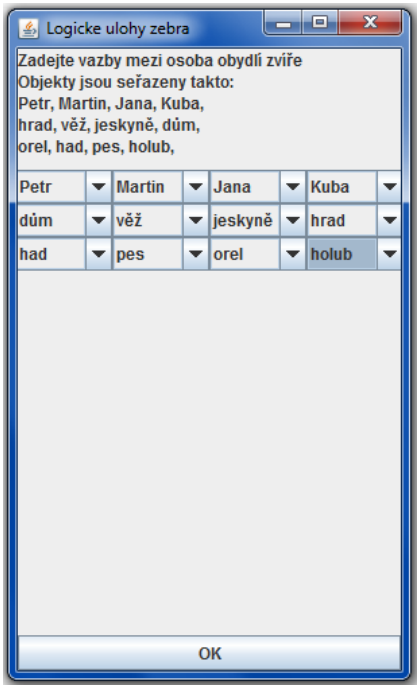
V dalším okně je třeba zadat relace mezi objekty, čímž uživatel definuje výsledek úlohy. Vazbu mezi objekty provede uživatel tím, že do jednoho sloupce vybere ty objekty, které chce spojit.

První řádek rozbalovacích seznamů je upraven tak, že nelze vybrat jiné než stanovené objekty.

Na následujícím obrázku okna je vidět, jak je zadána vazba mezi Petrem, domem a hadem. To by se dalo interpretovat jako:

- má (Petr, dům) – „Petr má dům.“
- má (Petr, had) – „ Petr má hada.“
- má (dům, had) – „Dům má hada.“

Obrázek 16:Zadání relací mezi objekty (Zdroj: vlastní)



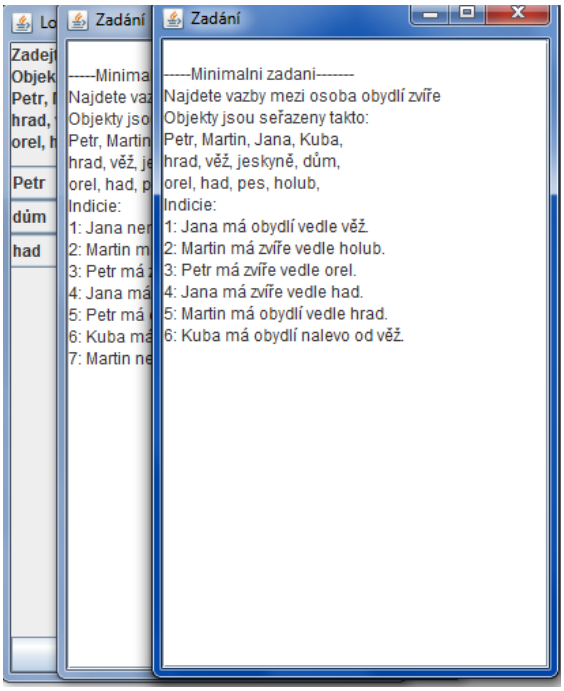
Jakmile je uživatel hotov s přiřazením relací mezi objekty, může stisknout potvrzovací tlačítko, které spustí algoritmy pro generování náhodného zadání pro zadanou úlohu.

Na dalším obrázku vidíme, že když opakovaně stiskneme tlačítko OK, aplikace vždy znovu vygeneruje nové náhodné zadání pro zadanou úlohu.

Zadání začíná krátkými instrukcemi a potom obsahuje minimální počet indicií pro vyřešení úlohy. Někdy se podaří vygenerovat menší počet indicií, což je způsobeno menším výskytem indicií, které podávají méně informací.

Text na textové ploše lze tahem myši označit a stiskem kombinace kláves Ctrl + C zkopírovat do textového editoru, do e-mailu nebo šířit jiným způsobem.

Obrázek 17: Opakovaně vygenerované zadání (Zdroj: vlastní)



7. Závěr

Hlavním cílem této práce byla vytvoření algoritmů generátoru úloh typu zebra a jejich následná implementace. Tento úkol zahrnoval nejprve letmé seznámení s terminologií a pravidly matematické logiky a některými pojmy teorie grafů a specifickými druhy reprezentace dat pohledem teorie grafů. Tyto teoretické poznatky byly posléze použity při popisu úlohy zebra, algoritmům jejího řešení a generování zadání k různým řešením. Zejména reprezentace dat z teorie grafů se osvědčila i v praktické části práce, v programovém řešení.

Cíl práce byl splněn naprogramováním funkční aplikace generátoru pro úlohy typu zebra na základě poznatků zjištěných v teoretické části práce.

Praktický výstup práce v podobě spustitelného programu byl koncipován jako poloautomatický generátor zadání pro úlohy zebra. Uživatel je totiž po spuštění aplikace umožněno definovat úlohu a její výsledné řešení, načež se teprve budou generovat různá zadání. Je tedy možné takto vytvořit nepřeberné množství různě těžkých úloh. Složitost úlohy uživatel může ovlivnit právě počátečním definováním úlohy. S větším množstvím objektů a kategorií a například i složitějším pojmenováním objektů roste obtížnost.

Jako programovací jazyk byl zvolen objektově orientovaný jazyk Java, protože jde o současný a často používaný programovací jazyk, a jeho kód je spustitelný na různých platformách. Tento programovací jazyk autorovi umožnil vytvořit skupinu přehledných zdrojových tříd, které tvoří celý program a také bylo možné z přeprogramovaných komponent sestavit přehledné jednoduché grafické uživatelské rozhraní, které umožnilo vnitřním algoritmům jejich plné využití uživatelem.

Práci je možné dále rozšířit. Mohou být definovány složitější možnosti zadání pro generování rozmanitějších úloh. A dále by pokračovatel mohl rozšířit aplikaci o herní část, která by umožnila uživateli přepnout z části generátoru úloh na herní rozhraní, kde by mohl jako hráč řešit dříve vygenerované úlohy.

Tato práce je přínosem zejména protože dokáže generovat libovolné množství různých úloh typu zebra, čímž se může stát zdrojem pobavení uživatelů navštěvující některé webové stránky, kde se můžou úlohy pravidelně obměňovat.

8. Seznam literatury

HEROUT, P. *Učebnice jazyka Java*, Kopp, 2000, ISBN 987-80-7232-323-4

KOTLÁN, I., KOTLÁN, P., VITTOVÁ, K. *Testy studijních předpokladů a základy logiky.* 5. vydání, Brno: Institut vzdělávání Sokrates s.r.o., 2006, 199 s., ISBN 80-89572-23-4

SPELL, B. *Java: Programujeme profesionálně,* Computer Press, 2002, 1040 s, ISBN 8072266675

Metodika k vypracování bakalářské / diplomové práce. EGER, L. [online] Aktualizace 1. 3. 2010, Dostupné na www: <ww.fek.zcu.cz>

Zebra [online]. Sisma, V. Aktualizace 10. 1. 2007 [cit. 12. 2. 2012]. Dostupné na www: <http://www.ms.mff.cuni.cz/sismv8am/cesky/zebra/zebra.html.en.iso.8859-5>

Teorie grafů a diskrétní optimalizace 1, RYJÁČEK, Z. [online]. Aktualizace 3. 1. 2007 [cit. 20. 2. 2012] Dostupné na www: <http://www.kma.zcu.cz/TGD1>

Creating Logic Puzzles, PEINTER, B. [online]. Aktualizace 14. 4. 2001 [cit. 15. 2. 2012] Dostupné na www: <http://www.mysterymaster.com/links/592FinalReport.pdf>

Java API [online]. Oracle, Dostupné na www: http://docs.oracle.com/javase/6/docs/api/

9. Seznam tabulek

10. Seznam obrázků

Abstrakt

ČERNOHOUZ, V. *Návrh a implementace výukové hry typu zebra*. Diplomová práce. Plzeň: Fakulta ekonomická ZČU v Plzni, 67 s., 2012

Klíčová slova: Logika, Einsteinova úloha, Zebra, Hlavolam, Programování, Java

Předložená práce se soustřeďuje na implementaci aplikace generující zadání k logickým úlohám typu zebra. Hlavní náplní vytvoření takového software je definování typů indicií (predikátů), algoritmů pro generování všech indicií k danému řešení a minimální množiny indicií postačující k dosažení řešení. Toto řešení bylo předem definováno uživatelem. Pro definování tohoto řešení uživatelem aplikace zahrnuje jednoduché grafické uživatelské rozhraní umožňující zadání úlohy a jejího řešení v několika krocích.

Program je dále rozšiřitelný. Mohou být definovány a přidány další typy indicií pro různější a rozmanitější vygenerované úloh. A dále vytvoření herní verze aplikace, která by umožnila řešit hráči úlohy vygenerované původní částí aplikace.

Abstract

ČERNOHOUZ, V. *Creation of Einstein's puzzle educational game*. Diploma Thesis. Pilsen: Faculty of Economics ZČU in Pilsen, 67 p., 2012

Key words: Logic, Einstein, Zebra, Puzzle, Programming, Java

This work aims to creation of application which generates set of hints to Einstein's logic game. The core of the creation of such software lies in definition of types of hints (predicates), algorithms which generates the set of all possible hints to a solution, minimal set of hints to a solution. Before generating these sets the solution is defined by the user. For this definition the application includes simple graphical user interface which allows user to define problem and its solution in several steps.

The work is extendable further. Other types of hints could be defined and added to make generated problems more diverse and various. And the second is extension of the application with a game environment for users who want to play Einstein's problems generated by the former part of application.