VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky
a informatiky

DIPLOMOVÁ PRÁCE

2010 Petr Oborný

VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra aplikované matematiky

Paralelizace, modifikace a implementace DLX algoritmu

Parallelization, modification and implementation of the DLX algorithm

2010 Petr Oborný

Poděkování

Chtěl bych poděkovat vedoucímu diplomové práce Mgr. Petrovi Kovářovi, Ph. D. za výborné vedení, cenné poznámky, podněty a připomínky, mnohé korektury a velkou ochotu.

Prohlášení

Prohlašuji, že jsem diplomovou práci na téma „Paralelizace, modifikace a implementace DLX algoritmu“ vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne: ..... Podpis: .....

Abstrakt

Diplomová práce podrobně popisuje Algoritmus X, který hledá řešení problému úplného pokrytí, a Dan-cing Links algoritmus (DLX).

Tato práce je zaměřena především na rozklady kopmletních grafů na předepsané podgrafy a tvorbou algoritmů k sestavení této úlohy a paralelnímu výpočtu jejího řešení. Nalezená řešení slouží jako základ pro již dokázaná tvrzení, umožňující zkonstruovat rozklady grafů vyšších řádů.

Dále se zde objeví dva způsoby převodu problému hledání řešení skládačky Eternity II na úlohu vhodnou pro Dancing Links algoritmus. Eternity II je zajímavý problém, který si jistě zasluhuje více pozornosti nejen proto, že je spojený s vysokou výhrou 2 000 000 USD za jeho vyřešení.

Klíčová slova

Dancing links, DLX, DLX algoritmus, Rozklad kompletního grafu, Eternity II

Abstrakt

This diploma thesis describes in detail the Algorithm X, that seeks solution of the exact cover problem, and the Dancing Links algorithm (DLX) with his modified version.

The focus of this is on complete graph decomposition into subgraphs and creation of algorithms to build the task and its parallel solution. It serves as a basis for already proved theorems, allowing to construct of higher orders graphs decompositions.

Furthermore, there are two ways given translation of the problem of finding the Eternity II puzzle solution into a problem solvable by the Dancing Links algorithm. Eternity II is an interesting problem that certainly deserves more attention not only because it is associated with a prize of 2 000 000 USD for its first solution.

Klíčová slova

Dancing links, DLX, DLX algorithm, Complete graph decomposition, Eternity II

Obsah

1 Úvod

Rozklad kompletního grafu na izomorfní stromy patří k jednomu ze známých a studovaných problémů teorie grafů. Existují dokázaná tvrzení, která umožní zkonstruovat na základě matematické indukce rozklady grafů vyšších řádů a proto je zapotřebí takové konstrukce najít i pro ty malé „startovací“ grafy, jakožto základ indukčního kroku. Ne všechny rozklady grafů různých řádů, i rozklady na jiné kostry než dvojhězdy, lze uhodnout, potom je potřeba je najít za pomocí výpočetní techniky. Rozklady se zpracovávají na počítačích také proto, že zjistit všechna řešení, nebo dokonce tvrdit, že žádný rozklad daného grafu neexistuje, je pro ruční počítání příliš obtížné, nebo dokonce nemožné.

Další problém, který jsme v této práci řešili je Eternity II skládačka, která je spojena s odměnou 2 000 000 USD pro prvního úspěšného řešitele. Eternity II byla na trh uvedena 28. července 2007 a to, že ani po necelých třech letech není vyřešena jen dokazuje jak náročné je najít její řešení. Její tvůrce matematik Christopher Monckton si tak velké náročnosti byl jistě vědom a proto se odvážil vypsat tak vysokou odměnu za její vyřešení.

Samotný text bude rozdělen do následujících částí. Začneme definicí problému úplného pokrytí a jeho řešení. Vše ještě osvětlíme také na příkladech. Dále si podrobně popíšeme Algoritmus X, který řeší úlohu úplného pokrytí. A také Dancing Links algorimtus, účinnou techniku k provádění Algoritmu X i s jeho modifikovanou verzí.

Poté bude následovat stěžejní kapitola této práce, která se zabývá rozkladem kompletního grafu na předepsané podgrafy a převodem této úlohy na problém úplného pokrytí. Povíme si také o tom, jak se dá tato úloha paralelizovat a popíšeme vytvořený paralelní program, který úlohu umí vygenerenovat a vyřešit.

V poslední části se dozvíme více informací o skládačce Eternity II a o tom jak tento logický problém matematicky reprezentovat a vytvořit k němu model odpovídající úloze úplného pokrytí hned dvěma způsoby. Na závěr uvedeme stručný popis tříd, vytvořených právě pro zmíněné převody.

2 Problém úplného pokrytí

V informatice je problém úplného pokrytí formulován jako rozhodovací problém, který má zjistit zda úloha má řešení a pak ho i nalézt, nebo jinak dokázat, že úplné pokrytí neexistuje.

Problém úplného pokrytí je NP-úplný problém [11] zařazený do kategorie problémů s omezujícími podmínkami. Tuto úlohu můžeme vyjádřit například maticí incidence (výskytu), či bipartitním grafem. Oba dva způsoby si ukážeme na příkladu později (kapitola 2.3).

Jedním z algoritmů, který dokáže najít všechna řešení problému úplného pokrytí, je Algoritmus X, o kterém je tato práce. Mezi nejznámější úlohy úplného pokrytí patří například pokrytí obrazce dílky pentomina, nebo nalezení řešení Sudoku.

2.1 Formální definice

Mějme množinu $\mathcal{X}$ a systém jejich podmnožin $\mathcal{S}$, pak úplné pokrytí množiny $\mathcal{X}$ je takový systém $\mathcal{S}^*$ množin z $\mathcal{S}$, který splňuje následující podmínky:

1. Průnik jakýchkoliv dvou různých množin systému $\mathcal{S}^*$ je prázdná množina, tj. množiny systému $\mathcal{S}^*$ jsou po dvou disjunktní. Jinými slovy, každý prvek z množiny $\mathcal{X}$ je v $\mathcal{S}^*$ obsažen nanejvýš jednou.

$$\bigcap_{A_i \in \mathcal{S}^*} A_i = \emptyset$$

2. Sjednocení všech množin systému $\mathcal{S}^*$ je $\mathcal{X}$, tj. množiny v $\mathcal{S}^*$ pokrývají množinu $\mathcal{X}$. Jinými slovy, každý prvek ze systému $\mathcal{X}$ je v $\mathcal{S}^*$ obsažen alespoň jednou.

$$\bigcup_{A_i \in \mathcal{S}^*} A_i = \mathcal{X}$$

Stručně řečeno, každý prvek $\mathcal{X}$ je obsažen v právě jedné množině systému $\mathcal{S}^*$. Ekvivalentně lze říct, že úplné pokrytí množiny $\mathcal{X}$ je systém $\mathcal{S}^*$ množin z $\mathcal{S}$ takový, který tvoří rozklad $\mathcal{X}$.

Úplné pokrytí může obsahovat také prázdné množiny, v dalším výkladu se ale omezíme pouze na úplná pokrytí, která prázdné množiny neobsahují. Úplné pokrytí je vlastně nějaký rozklad množiny $\mathcal{X}$.

2.2 Příklad 1

Nechť $\mathcal{S} = \{T, U, V, W\}$ je systém podmnožin množiny $\mathcal{X} = \{1, 2, 3, 4\}$ takový, že:

- $T = \{\}$,
- $U = \{1, 3\}$,
- $V = \{2, 4\}$ a
- $W = \{2, 3\}$.

Pak systém $\mathcal{S}_1^* = \{U, V\}$ tvoří úplné pokrytí $\mathcal{X}$, protože množiny $U = \{1, 3\}$ a $V = \{2, 4\}$ jsou disjunktní a jejich sjednocení je rovno $\mathcal{X}$ ($U \cup v = \{1, 2, 3, 4\} = \mathcal{X}$).

Systém $\mathcal{S}_2 = \{T, U, V\}$ tvoří také úplné pokrytí $\mathcal{X}$. I když přidání množiny $T = \{\}$ stále zachovává požadované vlastnosti řešení (každá z dvojic řešení je disjunktní a sjednocení opět tvoří celou množinu $\mathcal{X}$), tak systém $\mathcal{S}_2$ nebudeme považovat za úplné pokrytí, protože jsme je omezili pouze na taková, která neobsahují prázdné množiny.

Například systém $\mathcal{S}_3 = \{V, W\}$ nemůže být úplným pokrytím množiny $\mathcal{X}$, protože průnik $V \cap W = \{2\}$ není prázdná množina. Navíc tento systém ani nepokrývá prvek $\{1\}$, neboť $U \cup v = \{2, 3, 4\} \neq \mathcal{X}$.

2.3 Příklad 2

Nechť $\mathcal{S} = \{A, B, C, D, E, F\}$ je systém množin množiny $\mathcal{X} = \{1, 2, 3, 4, 5, 6, 7\}$ takový, že:

- $A = \{1, 4, 7\}$,
- $B = \{1, 4\}$,
- $C = \{4, 5, 7\}$,
- $D = \{3, 5, 6\}$,
- $E = \{2, 3, 6, 7\}$ a
- $F = \{2, 7\}$.

Pak systém $\mathcal{S}^* = \{B, D, F\}$ je úplné pokrytí, protože každý prvek z $\mathcal{X}$ je obsažen v právě jedné z množin:

- $B = \{1, 4\}$,
- $D = \{3, 5, 6\}$ nebo
- $F = \{2, 7\}$.

Navíc je systém $\mathcal{S}^* = \{B, D, F\}$ jediným úplným pokrytím. Zdůvodnit to můžeme následujícími argumenty.

Jelikož je prvek $\{1\}$ obsažený pouze v množinách A a B, pak musí úplné pokrytí obsahovat buď množinu A nebo B, ale ne obě zároveň. Vybereme-li nejdříve do úplného pokrytí množinu A, pak už do něj nemůžeme přidat B, C, E ani F, protože každá z těchto množin má s A společný prvek. Do řešení tedy přidáme poslední zbývající množinu D. Toto řešení, ale není úplným pokrytím, neboť sjednocením jeho množin není celá množina $\mathcal{X}$, protože neobsahuje prvek $\{2\}$ ($A \cup D = \{1, 3, 4, 5, 6, 7\} \neq \mathcal{X}$). Můžeme říci, že nelze nalézt úplné pokrytí, které by obsahovalo množinu A.

Když tedy začneme úplné řešení hledat výběrem množiny B, pak do něj určitě nemůžeme zahrnout ani A ani C, protože mají s B společný prvek. A protože D je jediná zbývající množina obsahující prvek $\{5\}$, musí být součástí úplného pokrytí. Tím, že obsahuje D, už ale nemůže obsahovat E (společné prvky $\{3\}$ a $\{6\}$). Pak F je zbývající množina a systém $\mathcal{S}^* = \{B, D, F\}$ je opravdu úplným pokrytím.

2.3.1 Matice incidence

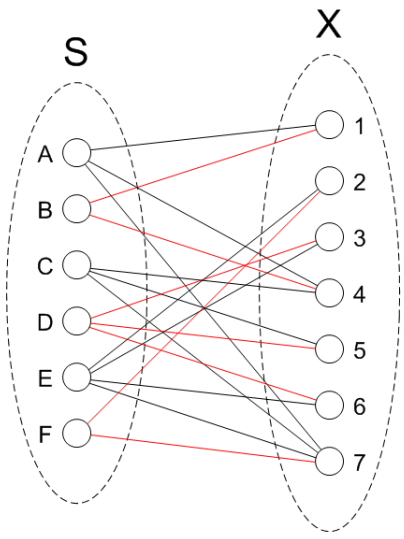
Ještě se podíváme jak vypadá zadaná úloha vyjádřená pomocí matice incidence. Řádky jsou množiny systému $\mathcal{S}$ a sloupce odpovídají prvkům množiny $\mathcal{X}$. Matice pak obsahuje jedničku, když množina systému $\mathcal{S}$ obsahuje prvek množiny $\mathcal{X}$.

	1	**2**	**3**	**4**	**5**	**6**	**7**
A	1	0	0	1	0	0	1
B	1	0	0	1	0	0	0
C	0	0	0	1	1	0	1
D	0	0	1	0	1	1	0
E	0	1	1	0	0	1	1
F	0	1	0	0	0	0	1

Úplné pokrytí je znázorněno červenou barvou.

2.3.2 Bipartitní graf

Stejná úloha reprezentovaná bipartitním grafem je na následujícím obrázku. Vrcholy S jsou množiny systému $\mathcal{S}$ a vrcholy X jsou prvky množiny $\mathcal{X}$. Hrana pak spojuje dva vrcholy, když množina systému $\mathcal{S}$ obsahuje prvek množiny $\mathcal{X}$.



Úplné pokrytí je opět znázorněno červenou barvou.

3 Algoritmus X

Algoritmus X je rekurzivní, nedeterministický algortimus, prohledávající nejdříve do hloubky, který najde všechna řešení úlohy úplného pokrytí kde $\mathcal{S}$ je systém podmnožin $\mathcal{X}$. V algoritmu reprezentujeme $\mathcal{S}$ maticí sestávající se z nul a jedniček. Cílem je vybrat podmnožinu řádků této matice tak, že číslice 1 se objeví v každém sloupci právě jednou.

Nedeterministický algoritmus je takový algoritmus, který v některých krocích může volit z několika možností dalších kroků. Nedeterministický algoritmus při stejném vstupu může dávat rozdílné výsledky.

Jeho opakem je deterministický algoritmus, který na stejný vstup (resp. Na stejné výchozí podmínky) reaguje vždy stejně (tedy předvídatelně) a v každém jeho kroku je vždy jednoznačně definován i krok následující.

3.1 Kroky algoritmu

Algoritmus X funguje takto:

1. Je-li matice A prázdná, tj. typu [0, 0], je problém vyřešen, algoritmus končí úspěšně.
2. V opačném případě vybereme sloupec c (deterministicky).
3. Vybereme řádek r takový, že $A_{r,c} = 1$ (nedeterministicky).
4. Zahrneme řádek r do částečného řešení.
5. Pro každý sloupec j takový, že $A_{r,j} = 1$,
   - pro každý řádek i takový, že $A_{i,j} = 1$,
     - vyřadíme řádek i z matice A,
   - vyřadíme sloupec j z matice A.
6. Opakujeme tento algoritmus rekurzivně od bodu 1. Na zredukované matici.

3.2 Poznámky k algoritmu

V takto stručně napsaném postupu je potřeba ještě vyjasnit některé kroky.

1. Může se stát, že je matice zadána takový způsobem, aby algoritmus vymazal všechny řádky a zároveň z matice nestihl vymazat všechny sloupce. Taková matice typu $[0, n]$ (0 - řádků, n - sloupců), vypovídá o tom, že některé sloupce zůstaly nepokryté a proto to není ta prázdná matice, pro kterou je problém vyřešen. Prázdná matice, vedoucí k řešení, tj. pokrytí každého sloupce řádkem obsahujícím jedničku, je matice typu **[0,0]**, tedy matice, která nemá žádný řádek ani sloupec.
2. S libovolným systematickým pravidlem pro volbu sloupce c najde tento algoritmus všechna řešení. Některá pravidla ale fungují lépe než ostatní. V některých úlohách, jak Knuth naznačuje, vede ke snížení počtu iterací modifikace výběru sloupce tak, aby vybraný sloupec měl co nejnižší počet jedniček [3].
3. Přidáme-li matici popis sloupců a řádků, můžeme pak o ní mluvit jako o tabulce.

3.3 Příklad

Jako příklad mějme následující úlohu. Příklad je převzat z wikipedie [6].

Problém pokrytí mějme zadán jako množinu k pokrytí $U = \{1, 2, 3, 4, 5, 6, 7\}$ a množinu řádků $\mathcal{S} = \{A, B, C, D, E, F\}$, kde:

- $A = \{1, 4, 7\}$,
- $B = \{1, 4\}$,
- $C = \{4, 5, 7\}$,
- $D = \{3, 5, 6\}$,
- $E = \{2, 3, 6, 7\}$,
- $F = \{2, 7\}$.

Tento problém je reprezentován tabulkou:

	1	**2**	**3**	**4**	**5**	**6**	**7**
A	1	0	0	1	0	0	1
B	1	0	0	1	0	0	0
C	0	0	0	1	1	0	1
D	0	0	1	0	1	1	0
E	0	1	1	0	0	1	1
F	0	1	0	0	0	0	1

Průběh Algoritmu X s Knuthovým heuristickým výběrem sloupce vypadá následovně:

Úroveň 0

- Krok 1 – tabulka není prázdná a algoritmus tedy pokračuje.
- Krok 2 – první sloupec zleva s nejmenším počtem jedniček je sloupec **1** (obsahuje dvě jedničky) a proto jej vybereme (deterministicky).

	1	**2**	**3**	**4**	**5**	**6**	**7**
A	1	0	0	1	0	0	1
B	1	0	0	1	0	0	0
C	0	0	0	1	1	0	1
D	0	0	1	0	1	1	0
E	0	1	1	0	0	1	1
F	0	1	0	0	0	0	1

- Krok 3 – oba řádky A a B obsahují jedničku ve sloupci **1** a proto je vybereme (nedeterministicky).
- Algoritmus se posune na úroveň 1 a prozkoumá jestli výběr řádku A vede k řešení.

  - **Úroveň 1: Výběr řádku** A
    - Krok 4 – přidáme řádek A do částečného řešení ($\mathcal{S}^* = \{A\}$).
    - Krok 5 – řádek A obsahuje jedničky ve sloupcích **1**, **4** a **7**:

	1	**2**	**3**	**4**	**5**	**6**	**7**
A	1	0	0	1	0	0	1
B	1	0	0	1	0	0	0
C	0	0	0	1	1	0	1
D	0	0	1	0	1	1	0
E	0	1	1	0	0	1	1
F	0	1	0	0	0	0	1

    - Sloupec **1** má jedničky v řádcích A a B; sloupec **4** v řádcích A, B a C; a sloupec **7** obsahuje jedničky na řádcích A, C, E a F. Proto tedy vyřadíme všechny zmiňované řádky – A, B, C, E a F a sloupce – **1**, **4** a **7** z matice:

	1	**2**	**3**	**4**	**5**	**6**	**7**
A	1	0	0	1	0	0	1
B	1	0	0	1	0	0	0
C	0	0	0	1	1	0	1
D	0	0	1	0	1	1	0
E	0	1	1	0	0	1	1
F	0	1	0	0	0	0	1

- Zůstal tak pouze řádek D a sloupce **2**, **3**, **5** a **6**:

	2	**3**	**5**	**6**
D	0	1	1	1

- Krok 1 – tabluka není prázdná a proto algoritmus pokračuje.
- Krok 2 – nejmenší počet jedniček v libovolném sloupci je 0 a sloupec **2** je první sloupec zleva s 0 jedničkami (deterministicky):

	2	**3**	**5**	**6**
D	0	1	1	1

- Proto tato větev algoritmu končí neúspěchem. Sloupec **2** už nelze pokrýt.
- Odebereme tedy řádek A z částečného řešení ($\mathcal{S}^* = \{\}$).
- A algoritmus nyní prozkoumá možnost výběru řádku B na první úrovni. Je proto nutné dostat tabulku do stavu, ve kterém byla před výběrem řádku A.

• **Úroveň 1: Výběr řádku** B

- Krok 4 – přidáme řádek B do částečného řešení ($\mathcal{S}^* = \{B\}$).
- Krok 5 – řádek B obsahuje jedničky ve sloupcích **1** a **4**:

	1	**2**	**3**	**4**	**5**	**6**	**7**
A	1	0	0	1	0	0	1
B	1	0	0	1	0	0	0
C	0	0	0	1	1	0	1
D	0	0	1	0	1	1	0
E	0	1	1	0	0	1	1
F	0	1	0	0	0	0	1

- Sloupec **1** má jedničky v řádcích A a B; sloupec **4** v řádcích A, B a C. Proto tedy z matice vyřadíme řádky A, B a C a sloupce **1** a **4**:

	1	**2**	**3**	**4**	**5**	**6**	**7**
A	1	0	0	1	0	0	1
B	1	0	0	1	0	0	0
C	0	0	0	1	1	0	1
D	0	0	1	0	1	1	0
E	0	1	1	0	0	1	1
F	0	1	0	0	0	0	1

- Zůstávají tak pouze řádky D, E a F a sloupce **2**, **3**, **5**, **6** a **7**:

	2	**3**	**5**	**6**	**7**
D	0	1	1	1	0
E	1	1	0	1	1
F	1	0	0	0	1

- Krok 1 – tabulka není prázdná a proto algoritmus pokračuje.
- Krok 2 – první sloupec zleva s nejmenším počtem jedniček (pouze jedna jednička) je sloupec **5** a proto jej vybereme (deterministicky):

	2	**3**	**5**	**6**	**7**
D	0	1	1	1	0
E	1	1	0	1	1
F	1	0	0	0	1

- Krok 3 – řádek D obsahuje jedničku ve sloupci **5** a proto jej vybereme (nedeterministicky).
- Algoritmus se posune na úroveň 2, kde bude moci prozkoumat pouze výběr řádku D.
    - **Úroveň 2: Výběr řádku** D
        - Krok 4 – přidáme řádek D do částečného řešení ($\mathcal{S}^* = \{A, D\}$).
        - Krok 5 – řádek D obsahuje jedničky ve sloupcích **3**, **5** a **6**:

	2	**3**	**5**	**6**	**7**
D	0	1	1	1	0
E	1	1	0	1	1
F	1	0	0	0	1

        - Sloupec **3** má jedničky v řádcích D a E; sloupec **5** pouze v řádku D; a sloupec **6** v řádcích D a E. Proto tedy z matice vyřadíme řádky D a E a sloupce **3**, **5** a **6**:

	2	**3**	**5**	**6**	**7**
D	0	1	1	1	0
E	1	1	0	1	1
F	1	0	0	0	1

        - Zůstal jen řádek F a sloupce **2** a **7**:

	2	**7**
F	1	1

        - Krok 1 – tabulka není prázdná a proto algoritmus pokračuje.
        - Krok 2 – první sloupec zleva s nejmenším počtem jedniček je sloupec **2**

a proto jej vybereme (deterministicky):

	2	**7**
F	1	1

- Krok 3 – řádek F obsahuje jedničku ve sloupci **2** a proto jej vybereme (nedeterministicky).
- Algoritmus se posune na úroveň 3, kde prozkoumá pouze výběr řádku F.
    * **Úroveň 3: Výběr řádku** F
        - Krok 4 – přidáme řádek F do částečného řešení ($\mathcal{S}^* = \{A, D, F\}$).
        - Krok 5 – řádek F obsahuje jedničky ve sloupcích **2** a **7**:

	2	**7**
F	1	1

        - Sloupec **2** má jedničku pouze v řádku F; a sloupec **7** také pouze v řádku F. Proto tedy z matice vyřadíme řádek F a sloupce **2** a **7**:

	2	**7**
F	1	1

        - Teď už nezůstal žádný řádek ani sloupec.
        - Krok 1 – tabulka je prázdná a proto algoritmus na této úrovni úspěšně našel řešení.
        - Jako řešení jsme tedy vybrali řádky B, D a F:

	1	**2**	**3**	**4**	**5**	**6**	**7**
B	1	0	0	1	0	0	0
D	0	0	1	0	1	1	0
F	0	1	0	0	0	0	1

        - Jinými slovy je tedy množina $\{B, D, F\}$ pokrytím zadané úlohy, právě proto, že každý sloupec je pokryt právě jednou jedničkou.
        - Není žádný další vybraný řádek pro úroveň 3 a proto algoritmus odebere z řešení řádek F ($\mathcal{S}^* = \{A, D\}$) a přesune se k výběru dalšího řádku na druhé úrovni.

- Není žádný další vybraný řádek pro úroveň 2 a proto algoritmus odebere z řešení řádek D ($\mathcal{S}^* = \{A\}$) a přesune se k výběru dalšího řádku na první úrovni.

- Není žádný další vybraný řádek pro úroveň 1 a proto algoritmus odebere z řešení řádek A ($\mathcal{S}^* = \{\}$) a přesune se k výběru dalšího řádku na úrovni 0.

- Není žádný další vybraný řádek pro úroveň 0 a proto algoritmus končí.

Algoritmus našel jediné řešení pokrytí a to $\mathcal{S}^* = \{A, D, F\}$. A jiné neexistuje.

4 Dancing Links

Dancing Links, známý také jako DLX, je technika, kterou navrhl Donald Knuth k účinnému provádění Algoritmu X. Mezi některé ze známých problémů pokrytí patří Sudoku či rozmístění dam na šachovnici [5].

Název Dancing Links je odvozen od způsobu jak algoritmus funguje. Každá iterace algoritmu způsobuje, že odkazy (links) „tančí“ se sousedními odkazy a vytváří tak „nádhernou taneční choreografii“. Hirosi Hitotumatu a Kohei Noshita vymysleli tento algoritmus v roce 1979 [1], ale až díky Donaldu Knuthovi se tento algoritmus zpopularizoval.

Problém úplného pokrytí je stále NP-úplná úloha, DLX je ale vhodný způsob jak ji implementovat.

4.1 Základní princip

Předpokládejme, že x je prvkem dvojitě linkovaného seznamu, kde $L[x]$ ukazuje na levého souseda a $R[x]$ na pravého. Potom operace:

$$L[R[x]] \leftarrow L[x],\ R[L[x]] \leftarrow R[x]$$

vymaže x ze seznamu. A operace:

$$L[R[x]] \leftarrow x,\ R[L[x]] \leftarrow x$$

vrátí prvek x zpátky na původní místo do seznamu.

Krása vrácení prvku zpět spočívá v tom uvědomit si, že stačí jen znát prvek x, respektive jeho položky $R[x]$ a $L[x]$. Dalším faktem je, že při zanořování do rekurze už není třeba vždycky znovu kopírovat celou úlohu, což zabírá spoustu výpočetního času i paměťového místa. Změny v ní nemají být permanentní, ale lze je vrátit zpět. Tím že algoritmus pracuje jen s odkazy se stává rychlejším.

Používání těchto operací způsobuje, že se ukazatelé prvků neustále mění a tím vytváří nádhernou taneční choreografii, odtud tedy Knuth tuto techniku pojmenoval jako techniku **tance odkazů (dancing links)**.

4.2 Čtyřsměrně prolinkované odkazy

Dobrým způsobem, jak reprezentovat systém podmnožin Algoritmu X, který tvořila matice A, je následující struktura. Každou jedničku v matici A bude představovat datový objekt x. Každý takový objekt x má 5 položek:

- $L[x]$ – odkaz na levého souseda,
- $R[x]$ – odkaz na pravého souseda,
- $U[x]$ – odkaz na horního souseda,
- $D[x]$ – odkaz na dolního souseda,
- $C[x]$ – odkaz na hlavičkový objekt.

Objekty řádků matice A jsou dvojitě linkované „do kruhu“ pomocí odkazů $L[x]$ a $R[x]$. Sloupce včetně hlavičky jsou pak dvojitě linkované „do kruhu“ pomocí odkazů $U[x]$ a $D[x]$. Každý sloupec obsahuje kromě prvků také *hlavičkový* objekt. Každý hlavičkový objekt h má navíc oproti datovému objektu x tyto položky:

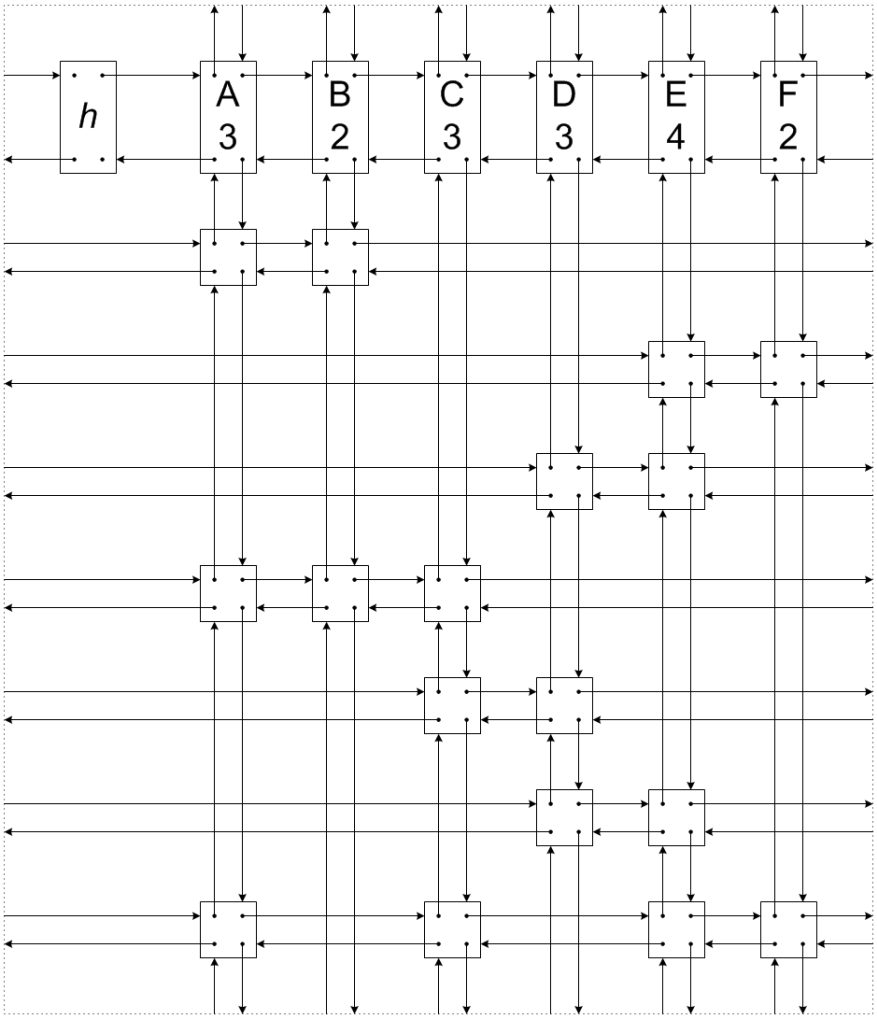
- $N[x]$ – název (jméno),
- $S[x]$ – velikost (počet jedniček).

Název slouží jako symbolické pojmenování sloupce, který hlavičkový objekt reprezentuje. A velikost je počet jedniček (objektů) v daném sloupci. Levé a pravé odkazy hlavičkových objektů spojují dohromady všechny hlavičkové objekty sloupců, které ještě zbývá pokrýt. Tento kruhový seznam navíc obsahuje speciální sloupcový objekt. Nazýváme jej *kořenový* objekt. Poskytuje nám seznam všech aktivních hlavičkových objektů. A ze všech svých položek využívá pouze levý a pravý odkaz, ostatní položky jsou pro něj nevyužité.

Jako příklad použijeme následující tabulku:

	A	**B**	**C**	**D**	**E**	**F**
1	1	1	0	0	0	0
2	0	0	0	0	1	1
3	0	0	0	1	1	0
4	1	1	1	0	0	0
5	0	0	1	1	0	0
6	0	0	0	1	1	0
7	1	0	1	0	1	1

Její čtyřsměrně prolinkovaná struktura vypadá takto:



Pro přehlednost jsme zde nezanesli odkaz $C[x]$, který vede od každého objektu k hlavičkovému objektu v témže sloupci.

Stejně tak jsme z důvodu přehlednosti zakreslili sousedy jdoucí vždy za sebou. Důležité je, aby odkazy jednotlivých objektů na každém řádku tvořily jediný cyklus, přičemž pořadí prvků v řádku není rozhodující. Obdobně je tomu i u sloupců.

Jednička v matici či tabulce pro nás bude i ve struktuře nadále jedničkou, která nyní bude v implementaci reprezentovaná datovým objektem.

4.3 Hledání řešení

Nedeterministický algoritmus pro nalezení všech úplných pokrytí se nyní dá převést na deterministický rekurzivní algoritmus „*search(k)*“ nad touto čtyřsměrně prolinkovanou strukturou a bude se spouštět s parametrem $k = 0$.

Algoritmus *search(k)* se skládá z následujících kroků:

- Je-li $R[h] = h$ pak vytiskneme řešení a vrátíme se o úroveň zpět.

- Jinak vybereme hlavičkový (sloupcový) objekt c.
- Zakryjeme sloupec c (zakrytí sloupce, viz níže).
- Pro každý řádek $r \leftarrow D[c], D[D[c]], ...$, dokud $r \neq c$,
  - nastavíme $O_k \leftarrow r$, zapamatujeme si tak aktuálně procházený řádek;
  - pro každý prvek $j \leftarrow R[r], R[R[r]], ...$, dokud $j \neq r$,
    - zakryjeme sloupec $C[j]$ (zakrytí sloupce, viz níže),
  - *search*$(k+1)$;
  - nastavíme $r \leftarrow O_k$ a $c \leftarrow C[r]$ (není třeba nastavovat pokud implementace rekurze nebude přepisovat ukazatele r a c),
  - pro každý prvek $j \leftarrow L[r], L[L[r]], ...$, dokud $j \neq r$,
    - vrátíme sloupec $C[j]$ zpět do struktury („odkryjeme“ sloupec $C[j]$).
- Vrátíme sloupec c zpět do struktury („odkryjeme“ sloupec c) a vrátíme se o úroveň zpět.

4.3.1 Tisk řešení

Tisk řešení je pak velmi jednoduchý. Provedeme jej tak, že vytiskneme řádky $O_0, O_1, ..., O_{k-1}$, kde máme zapamatované aktuálně procházené řádky, pro každé zanoření algoritmu, vedoucí k řešení. Tisk každého řádku řešení O_i, kde $i \in [0, k-1]$, pak opět zajistí jednoduché tisknutí posloupnosti $N[C[O_i]], N[C[R[O_i]]], N[C[R[R[O_i]]]]$, dokud $R[\cdots R[O_i]\cdots] \neq O_i$.

4.3.2 Výběr sloupce

Výběr hlavičkového objektu c můžeme provést buď velmi jednoduše tím, že zvolíme hned první nezakrytý sloupec ($c \leftarrow R[h]$). Nebo, pokud chceme minimalizovat procházení větvemi algoritmu, můžeme vybírat vždy ten sloupec, který obsahuje nejméně jedniček. Hlavičkový objekt c tedy vybereme takto:

- Nastavíme $c \leftarrow R[h]$,
- nastavíme $s \leftarrow S[R[h]]$
- pro každý $j \leftarrow R[R[h]], R[R[R[h]]], ...$, dokud $j \neq h$,
  - jestli $S[j] \leq s$,
    - nastavíme $c \leftarrow j$;
    - nastavíme $s \leftarrow S[j]$.

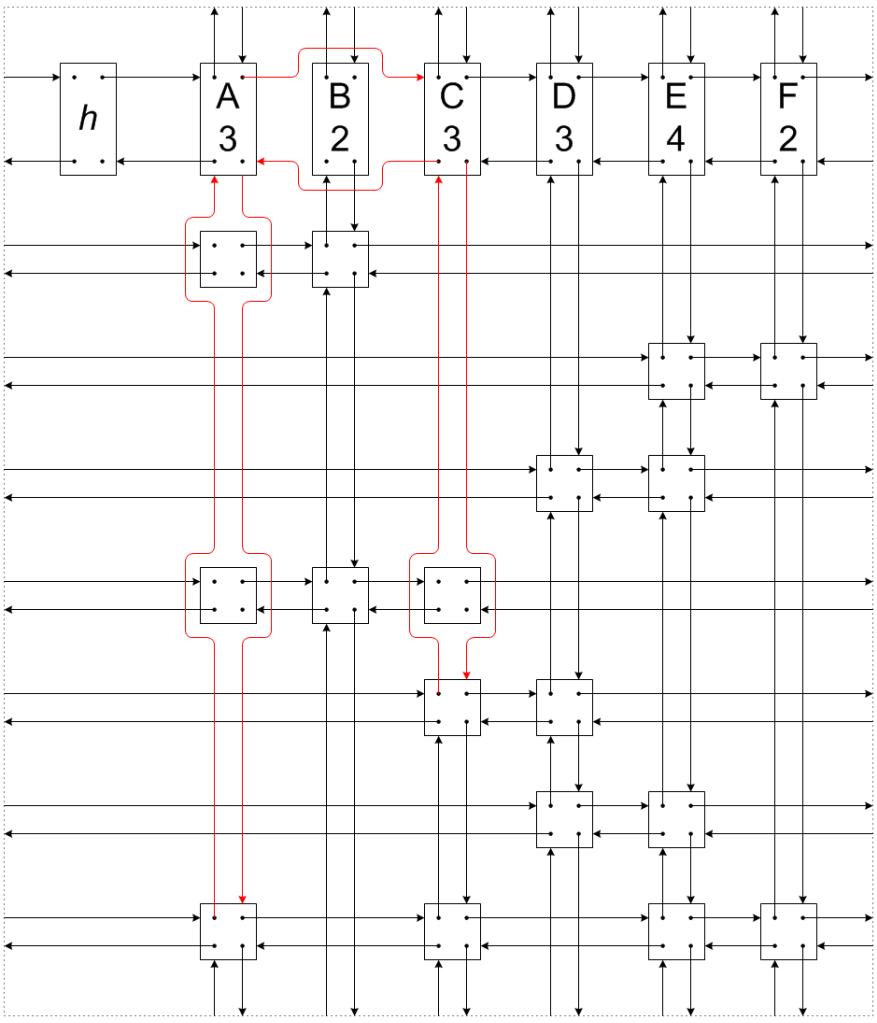
4.3.3 Zakrytí sloupce

Sloupec c (hlavičkový objekt) zakrýváme tak, že jej ze struktury odstraníme přepojením odkazů. A taktéž zakryjeme všechny řádky ze seznamu sloupce k zakrytí (řádky, které mají objekt nacházející se v pokrývaném sloupci).

- Nastavíme $L[R[c]] \leftarrow L[c]$,

- nastavíme $R[L[c]] \leftarrow R[c]$,
- pro každé $i \leftarrow D[c], D[D[c]], ...,$ dokud $i \neq c$,
  - pro každé $j \leftarrow R[i], R[R[i]], ...,$ dokud $j \neq i$,
    - nastavíme $U[D[j]] \leftarrow U[j]$,
    - nastavíme $D[U[j]] \leftarrow D[j]$,
    - nastavíme $S[C[j]] \leftarrow S[C[j]] - 1$.

Pro matici definovanou výše pak čtyřsměrně prolinkovaná struktura se zakrytým sloupcem B vypadá takto:



4.3.4 Vrácení sloupce zpět do struktury („odkrytí“ sloupce)

Sloupec c pak vrátíme zpět do struktury stejně jednoduše, jako když jsme jej zakrývali. Jen je třeba dát pozor na obrácené pořadí operací. Co bylo pokryto naposledy je třeba vrátit zpátky nejdříve. Je to způsobeno tím, že si zakryté objekty ponechají své odkazy beze změny a musí být navráceny opět na stejné místo. Takže když jsme zakrývali řádky zhora dolů, pak

je musíme obnovovat zdola nahoru. A objekty zakrývané zleva doprava obnovujeme zprava doleva.

- Pro každé $i \leftarrow U[c], U[U[c]], ...,$ dokud $i \neq c$,
  - pro každé $j \leftarrow L[i], L[L[i]], ...,$ dokud $j \neq i$,
    - nastavíme $S[C[j]] \leftarrow S[C[j]] + 1$,
    - nastavíme $U[D[j]] \leftarrow j$,
    - nastavíme $D[U[j]] \leftarrow j$,
- nastavíme $L[R[c]] \leftarrow c$,
- nastavíme $R[L[c]] \leftarrow c$.

Zakrývání a odkrývání zpět jsou právě ta místa kde říkáme, že odkazy tančí.

4.4 Modifikovaná verze

Může se stát, že si na řešení budeme potřebovat vynutit kromě samotného pokrytí i další podmínky. Například, pokud budeme chtít v úloze zakázat takové řešení, které by obsahovalo řádek A a zároveň řádek B, což není neřešitelný požadavek, stačí pouze poupravit DLX algoritmus.

Modifikovaná verze DLX algoritmu je taková, která rozeznává dva typy sloupců. Primární a sekundární. Primární, nebo též povinné sloupce, musí být pokryty právě jednou množinou systému podmnožin $\mathcal{S}$. A sekundární, nepovinné sloupce můžou být pokryty nejvýše jednou množinou systému podmnožin $\mathcal{S}$.

Algoritmus X musí mít pro tuto modifikovanou verzi upravenou kontrolu prázdnosti matice, tou bude taková, která bude mít pokryté všechny povinné sloupce, tou už ale nemusí být nutně matice typu [0, 0].

Vraťme se teď k již zmíněnému příkladu. Podmínku zákazu řešení obsahujícího jak řádek A, tak i B, již jednoduše zajistíme přidáním nepovinného sloupce do struktury. A oběma těmto řádkům doplníme jedničky pro tento nepovinný sloupec. Dojde-li potom k vybrání nejvýše jednoho z těchto řádků, druhý bude algoritmem vyřazen právě proto, že obsahuje jedničku v nepovinném sloupci.

5 Rozklad kompletního grafu

Ještě před samotným popisem rozkladu grafu si připomeneme několik pojmů z teorie grafů.

5.1 Definice

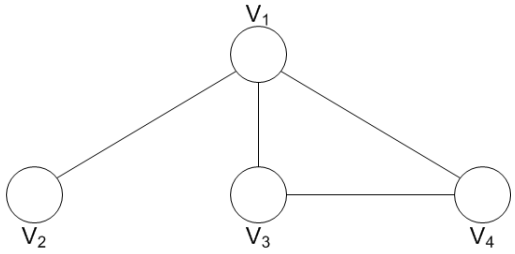
Grafy jsou jednou z mnoha významných struktur v diskrétní matematice. Nepleťme si je proto s grafem funkce!

Definice 1 ***Graf*** *(obyčejný či jednoduchý neorientovaný graf) je uspořádaná dvojice* $G = (V, E)$*, kde* V *je množina vrcholů a* E *je množina hran – množina vybraných dvouprvkových podmnožin množiny vrcholů.*

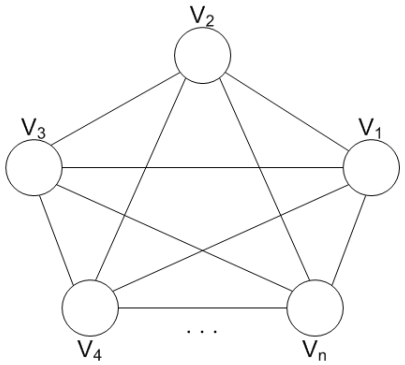
Hrana mezi vrcholy u a v je dvouprvková množina $\{u, v\}$, zkráceně píšeme uv. Vrcholy spojené hranou jsou sousední. Na množinu vrcholů známého grafu G odkazujeme jako na $V(G)$, na množinu hran $E(G)$.

Grafy se často zadávají přímo názorným obrázkem, jinak je lze také zadat výčtem vrcholů a výčtem hran. Například:

$$V = \{v_1, v_2, v_3, v_4\}, E = \Big\{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_3, v_4\}\Big\} = \{v_1v_2, v_1v_3, v_1v_4, v_3v_4\}$$



Kompletní graf K_n (též úplný graf) je takový graf na n vrcholech, který má všechny dvojice vrcholů spojené hranami. Nebo také můžeme říci, že je každá dvojice vrcholů u a v sousední. Pro $n \geq 2$ má kompletní graf $\binom{n}{2}$ hran.



Definice 2 ***Podgraf.*** *Podgrafem grafu* G *rozumíme libovolný graf* H *na podmnožině vrcholů* $V(H) \subseteq V(G)$*, který má za hrany libovolnou podmnožinu hran grafu* G *majících oba koncové vrcholy ve* $V(H)$*. Píšeme* $H \subseteq G$*.*

Podgraf je jednoduše řečeno část grafu.

Definice 3 ***Faktor grafu.*** *Podgraf* H *grafu* G *je faktor grafu* G*, jestliže množina vrcholů grafu* H *je totožná s množinou vrcholů grafu* G *(*$V(H) = V(G)$*).*

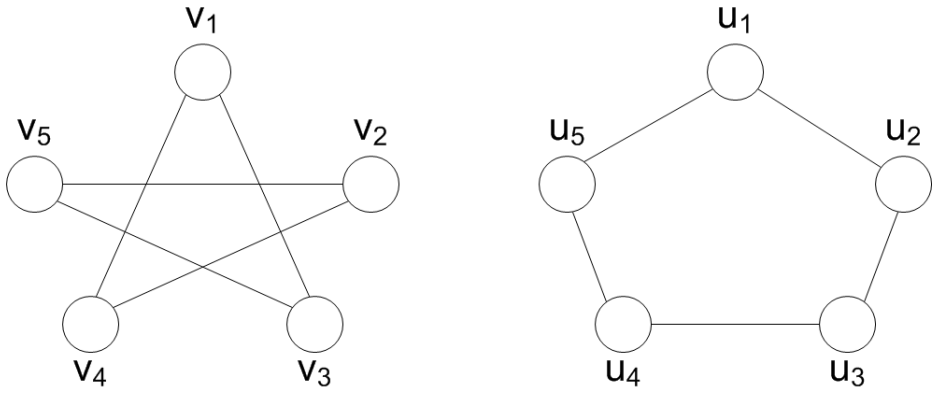
Faktor vznikne z grafu tak, že budeme odebírat pouze hrany nikoliv však vrcholy.

Definice 4 ***Izomorfizmus*** *grafů G a H je bijektivní (vzájemně jednoznačné) zobrazení $f : V(G) \to V(H)$, pro které platí, že každá dvojice vrcholů $u, v \in V(G)$ je spojená hranou v G právě tehdy, když je dvojice $f(u), f(v) \in V(H)$ spojená hranou v H.*

Grafy G a H jsou izomorfní pokud mezi nimi existuje izomorfismus. Píšeme $G \simeq H$.

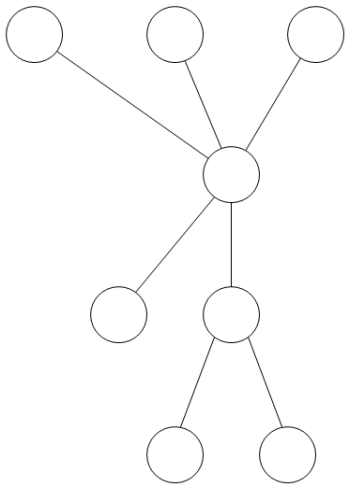
Izomorfizmus určuje „stejnost“ struktury grafu. Tento pojem byl zaveden proto, abychom mohli říci, že pro různá nakreslení zůstává graf stále stejný. Jako příklad nám postačí následující grafy, mezi kterými existuje izomorfizmus f.

$$\begin{aligned} f(u_1) &= v_1 \\ f(u_2) &= v_3 \\ f(u_3) &= v_5 \\ f(u_4) &= v_2 \\ f(u_5) &= v_4 \end{aligned}$$



Definice 5 ***Strom.*** *Graf T je stromem na n vrcholech pokud je to jednoduchý souvislý graf a zároveň obsahuje přesně $n - 1$ hran.*

Poznámka: Obvyklá definice říká, že strom je souvislý acyklický graf. Při pozdějším výkladu ale budeme spíše využívat vlastností výše definované neobvyklé verze.



Definice 6 ***Kostra grafu.*** *Kostrou souvislého grafu G je faktor v G, který je sám stromem.*

Kostra je speciální typ faktoru. Jinak lze též říci, že každá kostra je faktorem, obráceně to už ale neplatí.

5.2 Rozklad kompletního grafu

Rozklad kompletního grafu na izomorfní stromy patří k jednomu ze známých a studovaných problémů teorie grafů. Existují dokázaná tvrzení, která umožní zkonstruovat rozklady grafů řádu vyššího než 10, 12, 14, 16... A proto je zapotřebí prozkoumat ty „malé“ grafy. Ne všechny rozklady grafů různých řádů, i rozklady na jiné kostry než dvojhězdy, lze uhodnout, potom je potřeba je počítat za pomocí výpočetní techniky. Rozklady se zpracovávají na počítačích také proto, že zjistit všechna řešení, nebo dokonce tvrdit, že žádný rozklad daného grafu neexistuje, je pro ruční počítání příliš obtížné.

Jedním z úkolů této diplomové práce je rozkládat kompletní graf na n vrcholech na navzájem izomorfní kostry. Nalezení takového rozkladu, má potom smysl pouze pro grafy sudého řádu n.

Důkaz je triviální:

Kompletní graf má přesně $\binom{n}{2} = \frac{n!}{(n-2)! \cdot 2!} = \frac{1}{2} \cdot n \cdot (n-1)$ hran.

Kostra má $n-1$ hran.

Počet koster umístěných do grafu je vyjádřen podílem počtu hran kompletního grafu a počtu hran kostry: $\frac{\frac{1}{2} \cdot n \cdot (n-1)}{n-1} = \frac{n}{2}$.

Do grafu chceme umístit kostry celé a proto musí být podíl celočíselný, což je splněno pouze pro sudá n.

Teď již také přesně víme, na kolik koster budeme graf rozkládat a sice $\frac{n}{2}$.

5.3 Vytvoření úlohy pro Algoritmus X

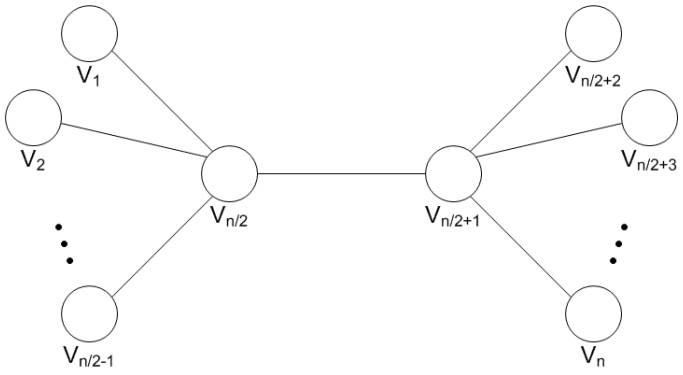
Pro Algoritmus X bude potřeba vytvořit množinu k pokrytí (sloupce matice) (X) a systém množin (řádky) (S), ze kterého bude vybírat řešení.

Potřebujeme pokrývat hrany grafu G, proto každé hraně grafu G bude odpovídat jeden sloupec, jehož hlavičkový objekt bude pojmenovaný tak, aby bylo rychle a jednoznačně určeno o jakou hranu v grafu se jedná. Například „v_1, v_2“. Hlavičkový řádek tabulky pak bude vypadat takto:

v_1, v_2	v_1, v_3	$\cdots$	v_1, v_n	v_2, v_3	$\cdots$	v_2, v_n	v_3, v_4	$\cdots$	v_{n-1}, v_n

S každým zvětšením řádu grafu bude sloupců mnohonásobně přibývat a název každého z vrcholů se mezi hlavičkovými objekty objeví hned $(n-1)$-krát a proto se můžeme i zde při samotném programování pokusit ušetřit nějaké místo a to například tak, že si vytvoříme objekty vrcholů jednoznačně pojmenované. Objekty hran pak budou odkazovat na dva vrcholy a sice počáteční a koncový vrchol hrany. Poté se každému hlavičkovému sloupci může přiřadit pouze odkaz na hranu, která už si své pojmenování dokáže poskládat sama. Tento způsob ušetří místo už při malých úlohách, natož pak na těch větších.

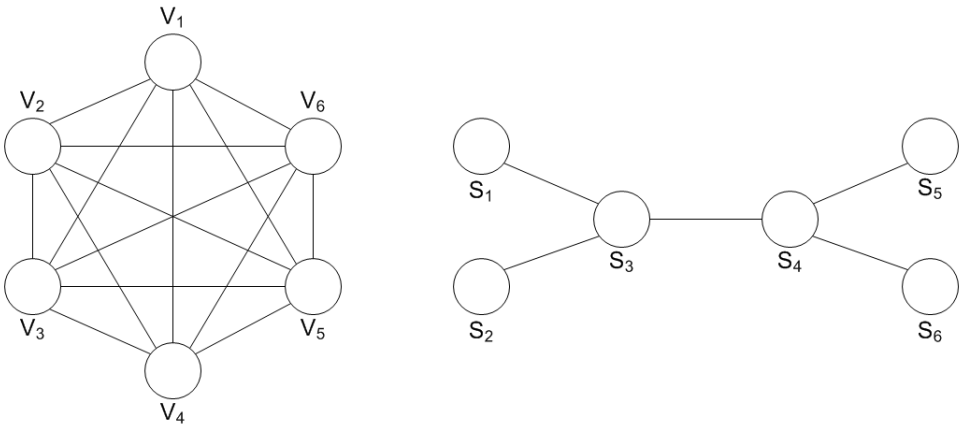
Řádky v tabulce musí přesně odpovídat požadované kostře, jejíž parametry musejí být dopředu známy. Nadále budeme pro jednoduchost a názornost pracovat s kostrami typu „dvojhvězda“. Je to takový graf, který má dva spojené centrální vrcholy, ze kterých vede stejný počet listů (viz obrázek).



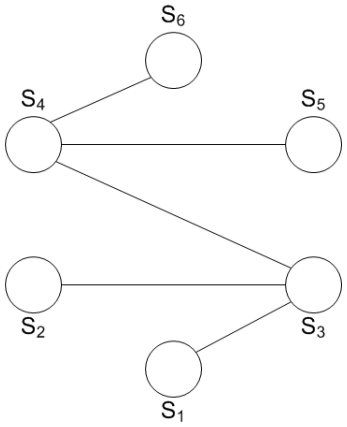
Pozorování: rozklad na dvojhvězdy je znám pro libovolné n. Použijí se jistá ohodnocení grafu pro konstrukci rozkladu, což je ale nad rámec tohoto textu.

Zásadním úkolem je vytvořit systém množin (S) k pokrytí množiny hran (X). Tedy nalézt všechna možná umístění kostry do úplného grafu. Můžeme například vybírat vždy ze všech hran právě $n-1$ a poté rozhodovat, zda tento výběr opravdu odpovídá tvaru zadané kostry.

Jako nejefektivnější metoda se ale jeví následující postup. Budeme procházet všechny permutace n-prvkové množiny všech vrcholů zadané kostry. Každá z těchto permutací bude určovat zobrazení vrcholů kostry na vrcholy kompletního grafu tak, že index prvku permutace bude určovat index vrcholu kompletního grafu. Mějme následují grafy řádu $n = 6$ (kompletní graf a jeho kostra – dvojhvězda).



Pak první permutace vrcholů kostry bude uspořádaná šestice: $[S_1, S_2, S_3, S_4, S_5, S_6]$, která říká, že vrchol S_1 se zobrazí na vrchol s indexem shodným s pořadovým číslem prvku této permutace a protože se vrchol S_1 vyskytuje na první pozici v permutaci, bude proto zobrazen na první vrchol kompletního grafu a tedy na V_1. Jako další příklad zvolíme novou permutaci a podíváme se jak vypadá její nakreslení. Takovou permutací bude například uspořádaná šestice $[S_6, S_4, S_2, S_1, S_3, S_5]$, kterou zobrazíme takto:



Když se zobrazí vrcholy kostry na vrcholy kompletního grafu, zobrazí se taktéž i hrany nadefinované kostry, protože rozkládáme kompletní graf, máme jistotu, že se zobrazí všechny hrany. Takto zvolená permutace tedy pokryla následující hrany původního kompletního grafu: $\{v_1, v_2\}$; $\{v_2, v_5\}$; $\{v_2, v_6\}$; $\{v_3, v_5\}$; $\{v_4, v_5\}$. Sestavit řádek v tabulce už pak nebude žádný problém, bude totiž obsahovat samé nuly ve všech sloupcích kromě těch, kde má obraz kostry hranu. Tam bude jednička.

	v_1, v_2	v_2, v_5	v_2, v_6	v_3, v_5	v_4, v_5
$[S_6, S_4, S_2, S_1, S_3, S_5]$	1	1	1	1	1

Pro přehlednost jsme vynechali všechny ostatní sloupce, protože obsahují hodnotu 0.

Najdeme-li všechny permutace, dostaneme tak všechna možná zakreslení kostry do kompletního grafu.

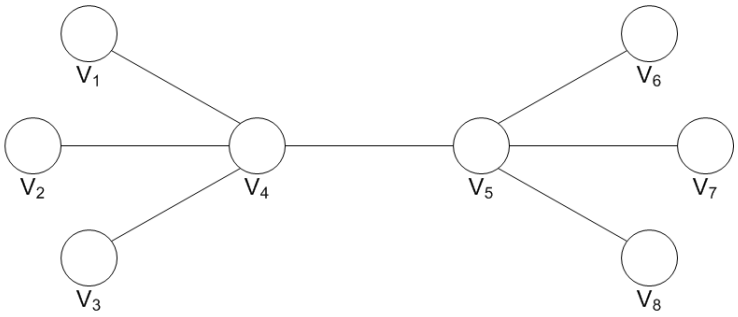
Aby úloha nepřekračovala svou velikostí neúnosnou mez, bude dobré nevytvářet stejné řádky, které vznikají když různá umístění kostry pokrývají stejnou množinu hran kompletního grafu. K už výše zvolené permutaci $[S_6, S_4, S_2, S_1, S_3, S_5]$, existuje dalších sedm, které pokryjí stejné hrany kompletního grafu a tedy nám vytvoří zbytečně se opakující řádky matice. Tyto opakující se permutace včetně zvolené vypadají takto:

- $[S_1, S_3, S_5, S_6, S_4, S_2]$,
- $[S_1, S_3, S_6, S_5, S_4, S_2]$,
- $[S_2, S_3, S_5, S_6, S_4, S_1]$,
- $[S_2, S_3, S_6, S_5, S_4, S_1]$,
- $[S_5, S_4, S_1, S_2, S_3, S_6]$,
- $[S_5, S_4, S_2, S_1, S_3, S_6]$,
- $[S_6, S_4, S_1, S_2, S_3, S_5]$,
- $[S_6, S_4, S_2, S_1, S_3, S_5]$.

Počet duplicitních řádků je vždy závislý na konkrétní definici kostry, kterou se bude pokrývat.

Takové duplicitní řádky by mohl program umět přímo nevytvářet, a tím úlohu zmenšit a urychlit tak nalezení jejího řešení, musí ale mít možnost je rozpoznat. Tu bude mít až když bude vědět, kdy jsou kostry izomorfní.

Nyní spočítáme kolik izomorfismů má dvojhvězda. Vzorce budeme odvozovat na dvojhvězdě, která má celkem osm vrcholů.



Pokud se mezi sebou prohodí první, druhý nebo třetí vrchol, bude to vždy ta samá kostra. Těchto výměn existuje přesně $P(3) = 3! = 6$. Obecně jde tedy o permutace nad polovinou vrcholů bez jednoho, ke kterému jsou všechny sousední. Je jich tedy $P\left(\frac{n}{2} - 1\right) = \left(\frac{n}{2} - 1\right)!$. Ke každé této permutaci můžeme permutovat i druhou stranu dvojhvězdy stejným způsobem.

Dostaneme tedy $\left(\left(\frac{n}{2}-1\right)!\right)^2$ izomorfních zobrazení. A pak také kostra bude vypadat stále stejně, když v ní zaměníme mezi sebou oba její centrální vrcholy. V tomto případě to jsou vrcholy V_4 a V_5. Počet izomorfismů se tím pádem zdvojnásobí. Když vynásobíme všechny tyto možnosti, pak nám vyjde, že dvojhvězda s $\frac{n}{2}-1$ listy u každého centra má vždy

$$\left(\frac{n}{2}-1\right)!\cdot\left(\frac{n}{2}-1\right)!\cdot 2 = 2\cdot\left(\left(\frac{n}{2}-1\right)!\right)^2$$

izomorfismů. Jestliže je pak vždy tolik řádků stejných, pak těch neidentických bude

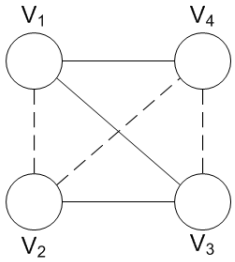
$$\frac{n!}{2\cdot\left(\left(\frac{n}{2}-1\right)!\right)^2}.$$

Jak efektivně a přitom stále obecně vynechat všechny izomorfismy si popíšeme až v kapitole 7.3.1 – Procházení všech permutací. Teď, když už víme jak sestavit úlohu, se zamyslíme nad samotných fungováním Algoritmu X. Ten si vybere řádek r (hrany kostry), který zahrne do řešení a pokryje všechny sloupce kde má řádek r hodnotu 1 (kde má kostra hranu). Pokrytí znamená odstranění všech ostatních řádků, které také mají hodnotu 1 v některém z pokrývaných sloupců. Což je ale přesně to co očekáváme, zahrneme-li totiž nějakou hranu do řešení, nechceme už pak nadále vybírat z řádků (koster), které tuto hranu obsahují také.

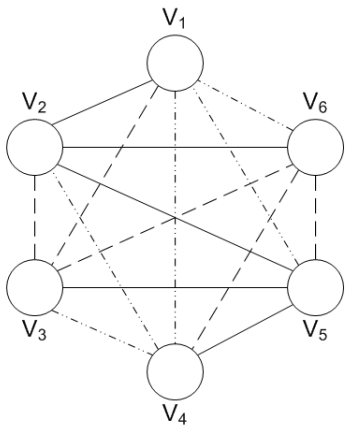
Pokryje-li algoritmus všechny sloupce, pak také pokryl všechny hrany kompletního grafu a nalezené řešení je tím co hledáme. Rozklad kompletního grafu pak z takového řešení vyčteme velmi jednoduše, a to pouze tak, že si přečteme jmenovku hlavičkového sloupce každého elementu řešení (hrany kostry). Každá tato jmenovka je totiž jednoznačně určena a přesně říká, kterou hranu každý element z řešení reprezentuje.

Ještě si pro názornost můžeme ukázat zmáné rozklady nejmenších grafů právě dvojhvězdou.

K_4



K_6



5.4 Výpočty

Bude zajímavé podívat se jak velká bude tato úloha v číslech. Bude nás zajímat

- **počet sloupců** je roven $\binom{n}{2} = \frac{n!}{(n-2)! \cdot 2!} = \frac{1}{2} \cdot n \cdot (n-1)$ (počet hran kompletního grafu),
- **počet řádků** je nejvýše $P(n) = n!$ (maximální počet permutací vrcholů kostry),
- **počet jedniček na řádku** je roven $n - 1$ (počet hran kostry).

Počet řádků může být menší než počet všech permutací, pokud zajistíme aby se v matici neopakovaly stejné řádky. Proto nás také bude zajímat počet řádků v matici bez duplicitních řádků pro dvojhvězdu jako rozkládající kostru,

- **počet neidentických řádků pro dvojhvězdu** je roven $\frac{n!}{2 \cdot \left(\left(\frac{n}{2} - 1\right)!\right)^2}$.

I když je ve vzorci pro počet neidentických řádků obsažen faktoriál, odhad nárustu velikosti úlohy pro dvojhvězdu je „pouze“ exponenciální a je větší než $\mathcal{O}(2^n)$ a zároveň menší než $\mathcal{O}(2, 1^n)$.

Podívejme se na tabulku vypočítaných hodnot.

	$n = 4$	$n = 6$	$n = 8$	$n = 10$	$n = 16$
počet sloupců	6	15	28	45	120
maximální počet řádků	24	720	40 320	3 628 800	20 922 789 888 000
počet jedniček na řádku	3	5	7	9	15
max. jedniček v úloze	432	54 000	7 902 720	1 469 664 000	37 661 021 798 400 000
počet neidentických řádků (pro dvojhvězdu)	12	90	560	3 150	411 840
počet jedniček v úloze (pro dvojhvězdu)	36	450	3 920	28 350	6 177 600

Je z ní okamžitě a na první pohled patrné, že bez možnosti určit izomorfismy bude úloha narůstat neúnosným způsobem. Pokud tuto možnost nebudeme dále uvažovat, nebo pokud budeme schopni identifikovat jen malý zlomek izomorfních koster, potom nejspíše nebude možné ani nagenerovat úlohu, protože se svým rozsahem nevejde do operační paměti žádného počítače. A právě tady musí nastoupit teoretický přístup.

6 Popis vytvořeného programu a jeho základních tříd

V této kapitole si vysvětlíme základní použité postupy, abychom se dozvěděli více o tom jak samotný program funguje.

6.1 Třída TaskCreator

Třída TaskCreator obsahuje pouze statické metody, které napomáhají vytvořit úlohu pro DancingLinksResolver. Velmi jednoduše vytváří čtyřsměrně prolinkovanou strukturu za pomocí těchto metod:

- **getRow(int rowLength)** – vytvoří jeden dvojitě prolinkovaný řádek;
- **getRoot(int count)** – vytvoří zadaný počet hlavičkových objektů a jeden kořenový objekt (který metoda vrací jako výstupní parametr) a dvojitě je prolinkuje;
- **getRoot(String...names)** – obdobně jako u **getRoot(int count)** jen s tím rozdílem, že vyplňujeme i pole pro název každého sloupcového objektu;
- **getRoot(List<String> names)** – obdobně jako u **getRoot(List<String> names)**, je zde pouze jiný způsob předání názvů sloupcových parametrů;
- **addRow(HeaderAndDataObject root, int... headerPositions)** – prolinkuje do struktury nový řádek, který obsahuje jedničky na zadaných pozicích (index začíná číslem 1, nultý je root);

- **addRow(HeaderAndDataObject root, TreeSet<Integer> headerPositions)** – obdobně jako u **addRow(HeaderAndDataObject root, int... headerPositions)**, je zde pouze jiný způsob předání pozic obsahujících jedničky.

Jako příklad vytvoříme čtyřsměrně prolinkovanou strukturu pro následující tabulku z příkladu v kapitole 3.3:

	1	**2**	**3**	**4**	**5**	**6**	**7**
A	1	0	0	1	0	0	1
B	1	0	0	1	0	0	0
C	0	0	0	1	1	0	1
D	0	0	1	0	1	1	0
E	0	1	1	0	0	1	1
F	0	1	0	0	0	0	1

Postup, jak takovou strukturu vytvořit, je jednoduchý:

```
// vytvorime root objekt
HeaderAndDataObject root = TaskCreator.getRoot("1", "2", "3", "4", "5", "6", "7");
// i tento zpusob je mozny
// HeaderAndDataObject root = TaskCreator.getRoot(7);

TaskCreator.addRow(root, 1, 4, 7);     // radek A
TaskCreator.addRow(root, 1, 4);        // radek B
TaskCreator.addRow(root, 4, 5, 7);     // radek C
TaskCreator.addRow(root, 3, 5, 6);     // radek D
TaskCreator.addRow(root, 2, 3, 6, 7); // radek E
TaskCreator.addRow(root, 2, 7);        // radek F
```

6.2 Třída DancingLinksResolver

Třída DancingLinksResolver představuje obecnou třídu, která implementuje Algoritmus X Dancing links technikou. Pracuje tedy s čtyřsměrně prolinkovanou strukturou, kterou převezme hned v konstruktoru třídy (parametr root).

Tato třída disponuje metodami:

- **searchMinHeader(int k)** – nalezení sloupce s nejmenším počtem jedniček;
- **coverColumn(HeaderAndDataObject header)** – metoda pro odebrání sloupce ze struktury;

- **uncoverColumn(HeaderAndDataObject header)** – metoda pro navrácení sloupce zpět do struktury;
- **findFirstSolution()** – nalezení prvního řešení;
- **findSolutions()** – nalezení všech řešení;
- **findSolutions(int numberOfSolutions)** – nalezení zadaného počtu řešení.

Podbrobný princip fungování těchto metod je popsán už v kapitole 4 – Dancing Links.

Máme-li již připravený root – kořenový objekt a tedy celou strukturu úlohy, řešení získáme pomocí několika málo příkazů:

```
// vytvoreni tridy, ktera umi resit ulohu
DancingLinksResolver dlx = new DancingLinksResolver(root);

// nalezeni a pruchod pres vsechna reseni
for (Solution sol : dlx.findSolutions()) {
    // tisk reseni
    System.out.println(sol);
}
```

6.3 Třída CompleteGraphTaskCreator

Tato třída je tou nejdůležitější. Rozkládá kompletní graf na předepsané podgrafy a vytváří tak úlohu pokrytí. Na základě zadaných parametrů:

- řád grafu,
- podgraf:
  - hrany podgrafu,
  - podmínky pro rozpoznání izomorismů,
- počet hledaných řešení,

vytvoří úlohu, kterou už je DancingLinksResolver schopen vyřešit. Jednotlivé parametry jsou důkladněji vysvětleny v kapitole 6.5 – Program a jeho nastavení.

6.3.1 Procházení všech permutací

Pro průchod všech permutací jsme použili následující algoritmus. Jako základ nám posloužil algoritmus, který je detailně popsán v [2] (kapitola 5.3).

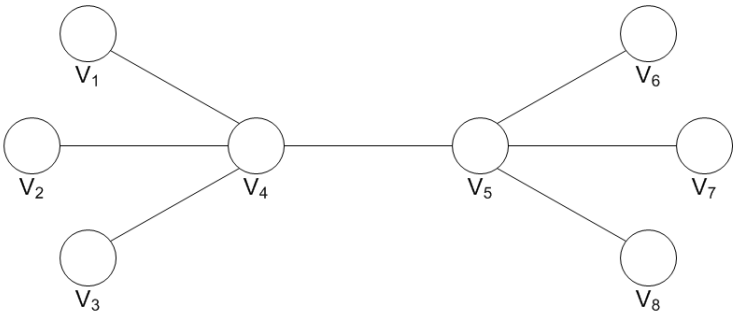
```
/**
 * Metoda pro pruchod vsech permutaci.
 *
 * @param subgraph - definice podgrafu,
 * @param root - ukazatel na korenovy objekt ulohy,
 * @param graphEdgeMap - mapa prirazujici kazde hrane jeji hlavickovy objekt,
 * @param graphNodes - seznam vrcholu grafu
 */
private void permutation(SubGraph subgraph, HeaderEdgeObject root,
        Map<Edge, HeaderEdgeObject> graphEdgeMap, Vector<Node> graphNodes) {
```

```
    int n = subgraph.getNodes().size(), i, j, select[] = new int[n];
    select[i = 0] = -1;
    While: while (i >= 0) {
        if (++select[i] >= n) {
            i--;
            continue;
        }
        for (j = 0; j < i; j++) {
            if (select[i] == select[j]) {
                continue While;
            }
        }
        if (++i < n) {
            select[i] = getStartValue(subgraph, select, i);
            continue;
        }
        processPermutation(select, subgraph, root, graphEdgeMap, graphNodes);
        i--;
    }
}


/**
 * Funkce, ktera vraci index od ktereho se bude pokracovat pri prohledavani ↩
     permutaci na zaklade podminek podgrafu.
 *
 * @param subgraph - definice podgrafu
 * @param select - pole permutaci
 * @param i - index v poli select, pro ktery se zjistuje pocatecni index
 * @return
 */
private int getStartValue(SubGraph subgraph, int[] select, int i) {
    for (Integer[] cond : subgraph.getConditions()) {
        if (cond[1] == i) {
            return select[cond[0]];
        }
    }

    return -1;
}
```

Je to nerekurzivní algoritmus a přitom stále obecný pro libovolný počet prvků n. Vytváří permutace naplňováním jednorozměrného pole *select*[] indexy 0 až $n-1$ postupně od prvku s indexem 0 až po prvek s indexem $n-1$. Navíc jsme tento algoritmus upravili tak, aby permutace generoval přímo podle zadaných podmínek izomorfismu (tvar těchto podmínek je uveden v kapitole 6.5.1 – Program a jeho nastavení, Server). Když se posune v poli a začne hledat další index, už to totiž nutně neznamená hledání od -1, ale podívá se jestli se na poslední naplněné místo v poli neváže podmínka izomorfismu. Pokud ano, je startovní index v posunutém místě pole nastaven na hodnotu stejnou jaká je na místě, se kterým je svázáno podmínkou izomorfismu.

Toto vylepšení, nejenom že kontroluje podmínky izomorfismu, a před samotným zpracováním permutace už tyto podmínky nemusíme znovu kontrolovat, ale navíc nám ušetří spoustu výpočetního času, protože mnoho kombinací poskládání pole *select*[] úplně přeskočí.

Příklad:



Podmínky pro takovýto případ jsou ve tvaru: $cond = \Big\{[1,2],[2,3],[4,5],[6,7],[7,8]\Big\}$. Uživatel tyto podmínky musí zadat sám, protože implementace automatického rozpoznání podmínek je nad rámec této práce.

Tyto podmínky říkají, že pokud zaměníme například vrchol 1 a vrchol 2, pak dostaneme opět stejný graf. Náš algoritmus si je ale nedokáže najít, pro každý zadaný graf je uživatel musí správně sestavit sám a předat je algoritmu jako vstupní parametr.

Algoritmus, který hledá permutace začne s prázdným polem $select[] = [,,,,,,,]$ a naplňuje jej zleva indexy vzestupně. Po několika cyklech se dostane do stavu, na kterém je jasně zřetelné uryhlení algoritmu. Pole bude naplněno následujícím způsobem $select[] = [4,5,6,7,,,,]$. Další, a tedy pátou, pozici nezačne procházet znovu od hodnoty 1, ale protože v *cond* existuje izomorfní podmínka pro pátou pozici $[4,5]$ vezme hodnotu na pozici 4 a hned za začátku nového cyklu k ní přičte jedničku ($select[4]+1 = 8$) a od této hodnoty teprve začne procházet pozici $select[5]$. Dostáváme tedy okamžitě pole $select[] = [4,5,6,7,8,,,]$. Pokud by podmínku nenašel, začal by pátou pozici procházet od hodnoty 1.

Seznam podmínek $cond = \Big\{[1,2],[2,3],[4,5],[6,7],[7,8]\Big\}$ tedy algoritmu říká, že pozice $select[2]$ se vždy začne procházet od hodnoty $select[1]+1$, dále pak $select[4] = select[3]+1, select[5] = select[4]+1, select[7] = select[6]+1$ a nakonec $select[8] = select[7]+1$.

Tímto postupem nagenerované permutace splňují vždy všechny stanovené podmínky izomorfismu.

6.4 Paralelizace

Pro paralelizaci problému jsme zvolili klient/server architekturu. Je tedy spuštěna jedna serverovská aplikace a neomezený počet klientských. Veškerá komunikace probíhá prostřednictvím TCP/IP přes java Sockety, přednastavené na port 5555. Toto nastavení je zaneseno napevno v kódu a uživatelsky se tedy nedá měnit (viz 7.5 – Program a jeho nastavení).

6.4.1 Server

Serverová část se stará o vyřešení problému jako celku. Je to tedy ta část, kde uživatel nastaví všechny potřebné parametry, jako jsou velikost úlohy, tvar podgrafu, počet řešení a další. Poté server spustí a ten si buďto nejdříve nageneruje úlohu, pokud tedy ještě nagenerovaná není, nebo přímo zadá vstupní soubor s úlohou. Poté čeká na připojení klientů a pro každého provede tento seznam akcí:

1. pošle celou úlohu,

2. pošle nastavení pro řešení úlohy (počet řešení, které ještě zbývají najít, popř. požadavek na výpis času či počtu řešení),
3. pošle „startovací řádek“,
4. přijímá řešení od klienta,
5. pokud už nalezl zadaný počet řešení, nebo pokud prozkoumal celou úlohu pak skončí, jinak se vrátí k bodu číslo 2.

Server má celou úlohu k dispozici a postupně si z její kopie odebírá řádky, které poslal jako startovací řádky a dostal na ně odpověď. Když už nezbývá žádný další řádek v seznamu. Pak je úloha kompletně prozkoumána a server může svou činnost ukončit.

6.4.2 Klient

Klientské části stačí nastavit pouze IP adresu serveru a spustit. Tímto se klientský program připojí k serveru a

1. přijme celou úlohu,
2. přijme nastavení pro řešení úlohy (zejména počet hledaných řešení),
3. přijme „startovací řádek“,
4. pokryje úlohu startovacím řádkem a začne hledat zbytek řešení,
5. pokud našel nějaké řešení, tak jej pošle na stranu serveru,
6. odkryje startovací řádek v úloze,
7. pokud dostane od serveru zprávu o dokončení úlohy, tak skončí, jinak se vrátí k bodu číslo 2.

6.5 Program a jeho nastavení

Pro náš program jsme zvolili programovací jazyk Java, byl vyvíjen ve verzi Java Standard Edition (Java SE) 1.6, update 16.

Samotný program je vygenerován do jar souboru (DLX.jar). V některých operačních systémech může být tento soubor samospustitelný. Main třída – main.Main (main – package, Main – třída) – je zadefinovaná v manifestu souboru. V tomto případě by se ale spustil bez užitku, protože má jeden povinný argument, aby se mohl rozhodnout jestli se má spustit jako server, či jako klient.

Pro úlohy vyššího řádu je také zapotřebí programu přidělit více operační paměti. Toto zajistíme opět jen přidáním dalších spouštěcích parametrů (-Xms$$$$m a -Xmx$$$$m, kde $$$$ je počet přidělených MB).

Veškerá programová komunikace mezi serverem a klienty probíhá pomocí TCP/IP na portu 5555. Pokud je již tento port obsazený, program nebude moci fungovat správně. Záměrně v celém programu není možnost číslo portu nastavit a to proto, že pokud se spustí server a klienti nebudou vědět na jakém portu server běží, tak nemají jinou možnost si to zjistit, než kontaktovat správce serveru. Což by bylo v rozporu s myšlenkou připojení neomezeného

počtu klientů.

Příklad správného spuštění programu vypadá následovně:

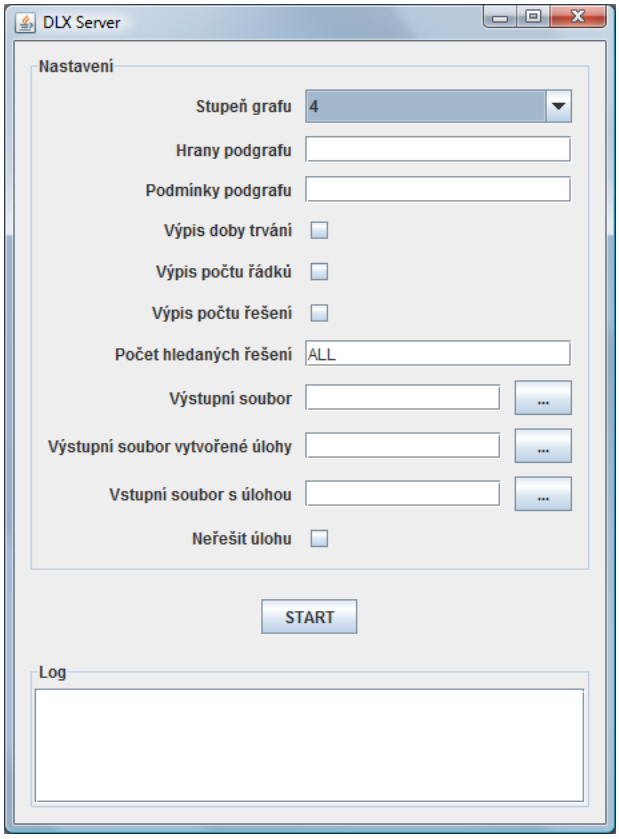
- Server (soubor: start_server.bat)
  `javaw -Xms512m -Xmx512m -jar DLX.jar main.Main -server`
- Klient (soubor: start_client.bat)
  `javaw -Xms512m -Xmx512m -jar DLX.jar main.Main -client`

6.5.1 Server

Po spuštění aplikace serveru je třeba do formuláře grafického uživatelského rozhranní (GUI) zadat několik důležitých údajů.

- **Řád grafu** – musí to být sudé číslo větší než nula a pro řád $n = 2$ je řešení triviální. Naopak pro velké řády n nemá smysl úlohu řešit, byla by pak natolik velká, že bychom se řešení nedočkali v rozumném čase. Navíc není nutné hledat rozklady tak velkých grafů, protože jako základ indukce dokázaných tvrzení o rozkladech grafu stačí tyto „malé“ grafy. Z těchto důvodu máme na výběr z předdefinovaného seznamu, který obsahuje řády 4 až 18 včetně.
- **Hrany podgrafu** – zde zadáme hrany podgrafu, kterým chceme úplný graf pokrývat. Hrany musí být v předepsaném tvaru: [(0,1),(1,2),...] (tj. hrana mezi 0. A 1. vrcholem a mezi 1. A 2. vrcholem,...) přičemž je povinná alespoň jedna hrana. Není-li parametr použit, a tedy políčko zůstane prázdné, pak bude program pracovat s přednastavenou kostrou – dvojhvězdou.
- **Podmínky podgrafu** – jsou tím myšleny podmínky izomorfismu zadané ve tvaru [(0,1),...] (tj. index 0. vrcholu podgrafu musí být větší než index 1. vrcholu podgrafu,...), taktéž je tím myšleno, že když zaměníme vrchol 0 s vrcholem 1 dostaneme opět tentýž graf.
- **Výpis doby trvání** – zaškrtneme pokud budeme chtít vypsat dobu trvání jednotlivých operací (doba generování úlohy, doby řešení jednotlivých bloků klientů, celková doba výpočtu všech řešení).
- **Výpis počtu řádků** – server vypíše kolik řádků obsahuje nagenerovaná úloha.
- **Výpis počtu řešení** – vypíše kolik řešení nalezli klienti v jednotlivých blocích a nakonec celkový počet nalezených řešení.
- **Počet hledaných řešení** – zde zadáme kolik nejvýše řešení má program hledat než skončí. Prázdné políčko, nebo text „ALL“ zajistí hledání všech řešení.
- **Výstupní soubor** – umístění a název souboru obsahujícího veškerý text z logu programu.
- **Výstupní soubor vytvořené úlohy** – soubor pro nagenerování úlohy, např. pro pozdější použití.

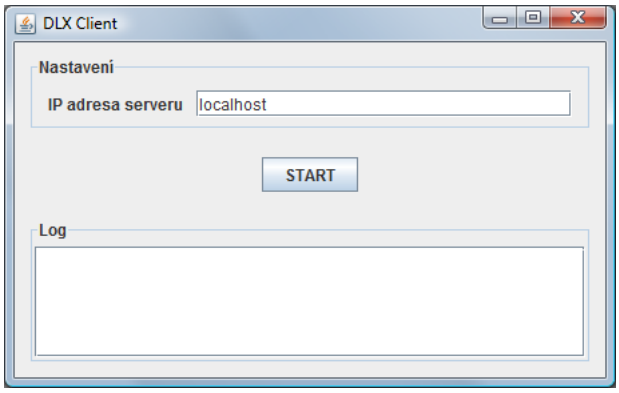
- **Vstupní soubor s úlohou** – cesta k souboru s nagenerovanou úlohou, při vyplnění této možnosti už není zapotřebí zadávat řád grafu, ani informace o podgrafu. Program si je sám načte ze vstupního souboru.
- **Neřešit úlohu** – zaškrtneme pokud nechceme úlohu řešit hned, ale úlohu pouze vygenerovat do souboru.
- **Start** – tlačítko pro zahájení výpočtu.
- **Log** – zde se zobrazují veškeré výpisy programu.



6.5.2 Klient

Klient má velmi jednoduché GUI. Disponuje pouze těmito ovládacími prvky:

- **IP adresa serveru** – zde se napíše IP adresa počítače, na kterém je server spuštěný.
- **Start** – tlačítko pro zahájení výpočtu.
- **Log** – zde se zobrazují veškeré výpisy programu.



7 Eternity puzzle

V této kapitole se budeme zabývat převodem úlohy Eternity II na vstup Algoritmu X. Nejdříve si ale krátce povíme i o první verzi této skládačky.

7.1 Eternity I

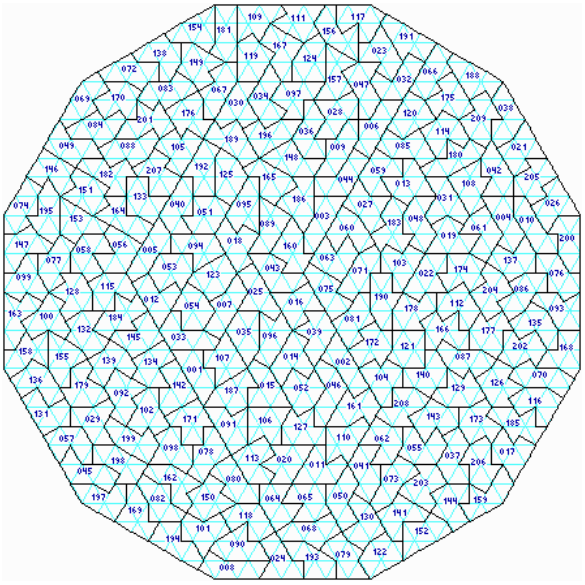
Eternity I (anglicky The Eternity puzzle) je geometrická skládačka, za kterou obdržel první úspěšný řešitel odměnu ve výši 1 000 000 britských liber. Tuto skládačku vytvořil matematik Christopher Monckton, který sám vložil polovinu peněz na výhru. Hra byla distribuována společností Ertl Company.

Christopher Walter Monckton, který se narodil 14. února 1952 není pouze matematik, ale také obchodní konzultant, politický poradce, spisovatel a mimo jiné vynálezce Eternity skládaček. Získal magisterský titul na univerzitě v Cardiff.

Samotná skládačka je tvořena 209 nepravidelnými dílky a po složení tvoří skoro pravidelný dvanáctiúhelník. Na trh byla hra uvedena v červnu 1999. Po svém uvedení se Eternity stala obrovským hitem a ve Velké Británii se stala nejprodávanější hrou, přestože její cena byla £35. Před uvedením Eternity na trh, předpokládal její tvůrce Monckton, že bude vyřešena za 1 - 3 roky.

Tuto nadmíru složitou skládačku vyřešili 15. května 2000, před prvním konečným termínem, dva matematici z Cambridge, Alex Selby a Oliver Riordan, kteří použili skutečně důvtipnou metodu k urychlení vyřešení této skládačky. Využili počítače, kterému předtím ulehčili práci tím, že se jim podařilo najít tzv. "end-game position" a špatné tvary(dílky) – wrong shape, které do sebe v žádném případě nemohly zapadnout.

Takto vypadá jedno z možných řešení této skládačky [8]:



7.2 Eternity II

Eternity II (anglicky The Eternity II puzzle) je skládačka, která je spojena s odměnou 2 000 000 USD pro prvního úspěšného řešitele. Eternity II byla na trh uvedena 28. července 2007 po velkém úspěchu předchozí skládačky Eternity I. Autorem Eternity II je opět Christopher Monckton, ale tentokrát hru vydala společnost TOMY UK Ltd.

Eternity II se skládá z 256 dílů, které se skládají na čtvercovou plochu 16 x 16 políček. Na 256 dílech jsou různé barevné vzory. Všech 256 čtverečků je rozděleno úhlopříčkami na čtyři stejné části, které jsou obarveny různou barvou. Barev je celkem 22 + šedá, kterou jsou označeny políčka, která musejí být po poskládání směrem k hranici. Tudíž je šedá barva jakousi barvou neutrální.

Na počátku je čtvercová plocha s 256 políčky a 256 dílky. Tyto dílky musí být sestaveny tak, aby strany celého obrazce byly šedé a barvy jednotlivých políček na sebe plynule navazovaly. Tudíž po složení vznikne obrazec, kde orámování čtverce bude šedé a uvnitř čtverce by měl vzniknout obrazec s plynulými přechody mezi dílky. Teoreticky nabízí Eternity II, na rozdíl od normálního puzzle, tisíce možností poskládání, matematicky by mělo těchto možností poskládání existovat přesně $256! \times 4^{256}$ [9].

Další výklad ale bude pokračovat na menší verzi této hry. Princip zůstává stále stejný, jen velikost hrací plochy bude tvořit pole velikosti 4 x 4 políčka a barev je v této verzi použito pouze 5 (4 + šedá). Takováto hra je k dispozici online na webových stránkách Eternity II [10] a slouží jako velmi chytlavá ukázka demonstrující samotnou hru, její principy, pravidla i obtížnost.

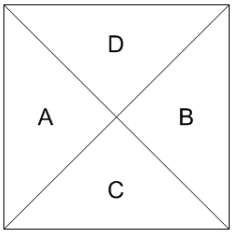
7.3 Vytvoření úlohy pro Algoritmus X

Aby se úloha Eternity skládačky dala řešit, musíme z problému skládání dílků vytvořit matematický problém. Pokusíme se tedy najít vhodnou matematickou reprezentaci tohoto problému, tak aby ji byl Algoritmus X schopný vyřešit.

Proto je právě tato kapitola, ve které si popíšeme hned dvě možnosti jak vytvořit úlohu pro Algoritmus X, jedním z hlavních přínosů této diplomové práce.

7.3.1 První způsob vytvoření úlohy

Stejně jako u rozkladu kompletního grafu budeme potřebovat definovat množinu $\mathcal{X}$ a systém jejich podmnožin $\mathcal{S}$. Ukažme si jeden z možných způsobů, jak množinu $\mathcal{X}$ reprezentovat. Prvky množiny $\mathcal{X}$ odpovídají sloupcům tabulky. Nejdříve, se podívejme jak jedna taková hrací kostka vypadá. Místo barev jednotlivých částí kostky jsou písmena A až D.



Kostek máme dohromady tolik, kolik je políček na hracím plánu. Na malém plánu to je 4 x 4 = 16. Úloha bude splněna ve chvíli kdy pokryjeme všech 16 sloupců, tj. použijeme všech 16 kostek. A proto základní část tabulky bude tvořena právě šestnácti sloupci, i-tá kostka má jedničku právě v i-tém sloupci.

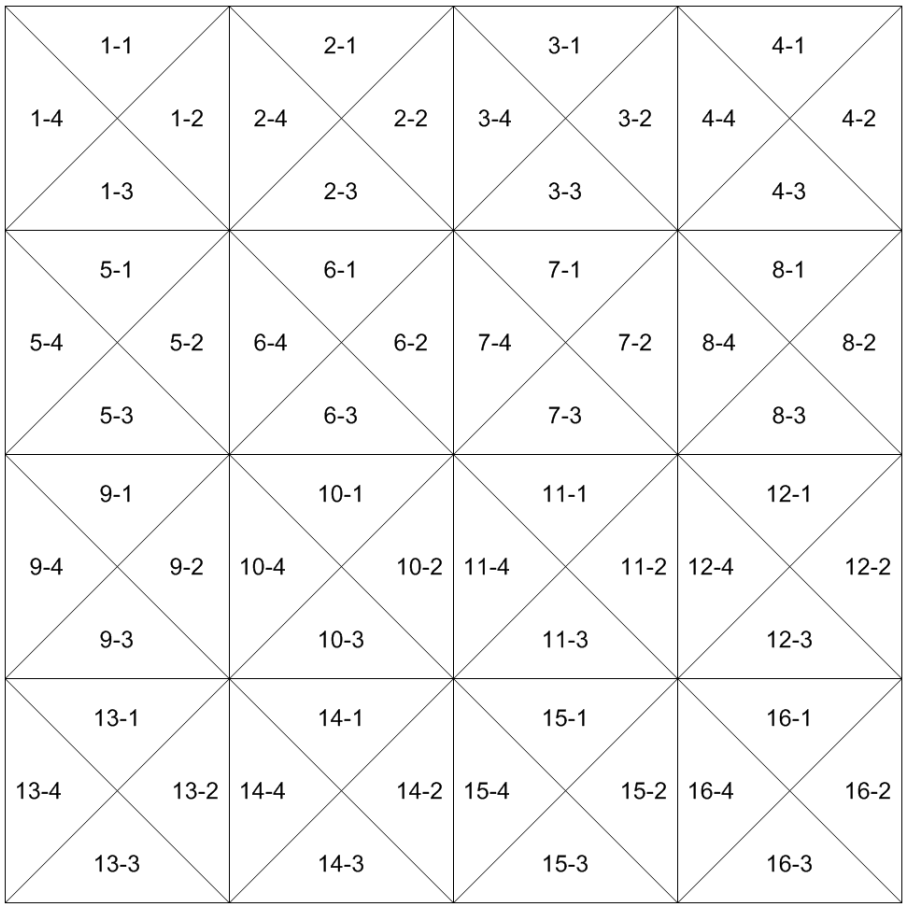
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
1. kostka	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2. kostka	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
⋮	0		⋱													0
16. kostka	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Ještě ale bude zapotřebí uvážit všechny možnosti na která pole je možno každou kostku položit a započítat všechny možnosti, jak ji otočit a tím vytvořit kompletní systém množin $\mathcal{S}$. Přibudou tak další sloupce, které nám pomohou s výběrem správného řešení. Počet řádků tedy bude mnohem větší, hodnoty se v této základní části tabulky mnohokrát zopakují. Lišit se budou až v další, nepovinně pokrývané, části tabulky (viz dále).

Nejdříve si ale očíslujeme políčka hrací plochy následujícím způsobem:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Každá kostka je rozdělena na 4 části, budeme tedy potřebovat ještě jemnější dělení:



Jak jsme si již řekli, úloha bude vyřešena, když pokryjeme všechna pole a použijeme tak všechny kostky. Navíc ale potřebujeme zajistit, aby algoritmus poznal, které řádky, možnosti umístění a natočení kostky, už k řešení nevedou. Takové podmínky jsme schopni vynutit přidáním nepovinných sloupců do struktury a vytvářet tak úlohu pro modifikovanou verzi Dancing Links algoritmu. Jen si připomeneme, že se jedná o sloupce, které k vyřešení problému nemusí být pokryty, hrají ovšem důležitou roli pro výběr řádků, které odpovídají přípustnému umístění kostek.

V každém z $16 \cdot 4 = 64$ dílků může být jiná barva. Proto jedno políčko hrací plochy může reprezentovat hned čtyřikrát počet barev, tedy 4 x 5 = 20, sloupců. Přiřadíme tedy barvám písmena A až E a podíváme se, jak bude vypadat hlavičkový řádek celé tabulky.

1	$\cdots$	**16**	**1-1A**	**1-1B**	$\cdots$	**1-1E**	**1-2A**	$\cdots$	**1-4E**	**2-1A**	$\cdots$	**16-4E**

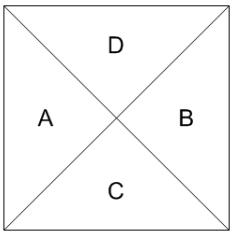
Například 2-1A znamená, že jde o první dílek kostky umístěné na druhém políčku hrací plochy, který má barvu A. 2-1A tedy přesně znamená 2-políčko, 1-dílek, A-barva.

Všechny tyto údaje budou sloužit pro přesné zjištění, kde kostka právě leží a jak je otočená.

Dále si v těchto sloupcích můžeme uchovávat informace následujícího typu. Pokud na dílek 1-2 připadne barva A, pak i na dílek 2-4 musíme umístit barvu A a tedy odstranit všechny

řádky úlohy, které mají na dílku 2-4 barvu jinou než A. Barvě A na dílku 1-2 proto přidělíme hodnotu 0, ostatní barvy na dílku 1-2 dostanou hodnotu 1. Sousednímu dílku 2-4 přidělíme hodnoty pro jednotlivé barvy přesně naopak, dílek 2-4 tak bude mít pouze jednu jedničku právě u barvy A. Toto rozmístění hodnot zapůsobí na úlohu přesně tak, jak očekáváme. Algoritmus bude muset zakrýt sloupec 2-4A, což znamená zakrýt všechny ostatní řádky, které taktéž obsahují jedničku ve sloupci 2-4A. Bystrý čtenář si již jistě uvědomil, že jedničku pro dílek 2-4 mají všechny barvy kromě té jediné, která má zůstat a sice barva A.

Zvolme si náhodný příklad obarvení kostky:



Řekněme, že tato kostka bude ležet na poli číslo **6**. Pak pro kostku, tak jak je na obrázku, a všem čtyřem otočením (ve směru hodinových ručiček) odpovídají následující řádky v tabulce:

	1	⋯	**16**	**2-3 D**	**5-2 A**	**6-1 ABCE**	**6-2 ACDE**	**6-3 ABDE**	**6-4 BCDE**	**7-4 B**	**10-1 C**
1. kostka 0°	1	0	0	1	1	1	1	1	1	1	1

	1	⋯	**16**	**2-3 A**	**5-2 C**	**6-1 BCDE**	**6-2 ABCE**	**6-3 ACDE**	**6-4 ABDE**	**7-4 D**	**10-1 B**
1. kostka 90°	1	0	0	1	1	1	1	1	1	1	1

	1	⋯	**16**	**2-3 C**	**5-2 B**	**6-1 ABDE**	**6-2 BCDE**	**6-3 ABCE**	**6-4 ACDE**	**7-4 A**	**10-1 D**
1. kostka 180°	1	0	0	1	1	1	1	1	1	1	1

	1	⋯	**16**	**2-3 B**	**5-2 D**	**6-1 ACDE**	**6-2 ABDE**	**6-3 BCDE**	**6-4 ABCE**	**7-4 C**	**10-1 A**
1. kostka 270°	1	0	0	1	1	1	1	1	1	1	1

Zápis 6-1ABCE znamená, že umístíme stejnou hodnotu do sloupců 6-1A, 6-1B, 6-1C a 6-1E. Dále jsme pro zjednodušení vypsali pouze ty nepovinné sloupce, kde bude zapsaná hodnota dílku rovna 1, všechny ostatní nepovinné sloupce mají hodnotu 0.

Celkem bude jeden řádek tabulky obsahovat maximálně, v případě, že není umístěn u kraje, přesně $1 + 4 \cdot (5 - 1) + 4 \cdot 1 = 1 + 4 \cdot 5 = 21$ jedniček (viz kapitola 6.4 Výpočty).

Tím že do řešení zahrneme **1. kostku 0°** se zajistí:

- pro povinnou část struktury – odstraňení řádků s jedničkou ve stejném sloupci. Znamená to tedy odstranění všech řádků, kde je uvažovaná kostka jen pootočená, a také všech řádků kde je právě tato kostka umístěná na jiná políčka;

- pro nepovinné sloupce – odstraňení všech nežádoucích barev sousedních políček. Například po dílku 2-3 budeme požadovat barvu D (vynutíme si ji jedničkou ve sloupci 2-3D). Všechny ostatní možnosti chceme vyřadit. Přesně to se ale také stane, tabulka má v řádcích s barvou D na dílku 2-3 ve sloupcích 2-3A, 2-3B, 2-3C a 2-3E hodnotu 1 a pouze ve 2-3D hodnotu 0, proto algoritmus nezakryje právě jen tyto řádky, ale všechny ostatní, které mají ve sloupci 2-3D hodnotu 1, což jsou všechny ostatní barvy, jenom ne D. Tím, jsme zajistili výběr pouze těch řádků, kde leží požadovaná barva.

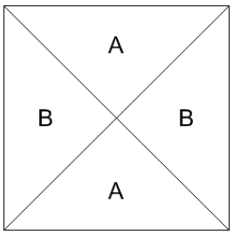
7.3.2 Poznámky k prvnímu způsobu

K takto nadefinované úloze je ještě zapotřebí zmínit pár dodatků či návrhů na zlepšení. Například jsme se doposud nezabývali podmínkou pro správné řešení úlohy, kterou je vynucení šedé barvy okolo okraje hrací plochy. Tohoto lze docílit hned dvěma způsoby, kde prvním by mohlo být přidání jednoho řádku do úlohy, který se hned stane součástí řešení. Takový řádek by pro všechny krajní dílky obsahoval hodnotu 1 ze všech nepovinných sloupců právě pro šedou barvu, to je $4 \cdot n$ jedniček. A do všech povinných sloupců dosadíme hodnotu 0, protože si pouze vynutíme barvu okraje, nepokryjeme však žádnou kostku. V našem případě je to přesně 16 jedniček, které zajistí správnou barvu okraje.

Druhým způsobem, jak docílit téhož je už spojený se samotnou implementací algoritmu generování řádků, do kterého by se musel vpravit kontrolní mechanismus, který by nepovolil položit šedou barvu nikam jinam než na okraj. Tato kontrola by mohla využívat poznatku předchozího způsobu. Kontrolovalo by se, jestli se nepokoušíme zapsat šedou barvu někam, kam to nemá povoleno. Taková úloha by byla hned od začátku mnohem řidší a tím by kladla menší nároky na místo v operační paměti.

Není špatný nápad předpokládat ušetření paměti tím, že nám stačí vynutit si barvu vždy jen pravého a spodního souseda. V úloze tak i nadále zůstanou podmínky pro každé rozhranní dvou sousedních políček a navíc ušetříme $8 \cdot n^4$ jedniček.

Další prostor pro úsporu paměti nabízí kontrola uspořádání barev na kostce. Vezměme si kostku, která má každé dvě protilehlé barvy stejné.



Pro takové obarvení má smysl otáčet kostku jen jednou o 90°, dalším otáčením, budeme dostávat už jen shodné řádky. Taková kontrola by pro každou jednu tuto kostku ignorovala $2 \cdot n^2$ řádků a pro kostku, která by byla celá z jedné barvy $3 \cdot n^2$ řádků.

Poslední, avšak jeden z nejdůležitějších postřehů, je interpretace řešení úlohy zpět na řešení hry. Bez možnosti převést řešení zpět by vůbec nemělo smysl ani řešení úlohy hledat. Převod je ovšem velmi jednoduchý. Povinné sloupce označují pořadové číslo kostky. Ze sloupce, u kterého má řádek hodnotu 1, vyčteme pořadové číslo. Z každého řádku řešení tedy okamžitě poznáme o jakou kostku se jedná a na hrací plochu ji umístíme podle toho, které nepovinné sloupce obsahují hodnotu 0. Nejdříve si připomeneme jak vypadá označení nepovinných sloupců. Například 6-1A znamená, že se jedná o šesté políčko hrací plochy, první dílek a barvu A. Označení čtyř z těchto sloupců musí vždy začínat stejným prefixem, kostka totiž vždy leží celá na jednom políčku. Tento prefix tedy udává číslo políčka. Samotné natočení kostky vyčteme z druhé části označení. Barva ležící na dílku 6-1 je právě ta, která se nenachází v řešení. To proto, že jsme ležící barvě přiřadili hodnotu 0. Takže buď správnou barvu zjistíme vylučovací metodou tak, že hledaná barva dílku 6-1 není žádná z 6-1x v řešení. Nebo pokud jsme se nerozhodli pro podmínky pouze pro pravého a spodního souseda, správnou barvu zjistíme u sousedního dílku, který má jedničku v řešení u barvy, kterou hledáme. Zároveň ale musí i ostatní barvy ležet přesně tam, kde to od nich řešení požaduje. Jejich umístění si obdobně přečtemě ze sloupců 6-2x, 6-3x, 6-4x.

Umístění kostky do hrací plochy by se dalo zjistit i jednodušeji, pokud bychom si zavedli ještě dalších n^2 sloupců, které by sloužily pro uchování této informace. Stejně tak jako máme pořadové čísla kostek, by nám přibyla pořadová čísla políček, na které se kostka právě umisťuje. Řádek by obsahoval jedničku mezi těmito sloupci právě tam, kde pořadové číslo políčka bude shodné s prefixem právě čtyř sloupců. Je ale potřeba zvážit jestli je to výhodné, protože samozřejmě dojde ke zvětšení úlohy, bez přidání jakékoliv informace, která by již v řešení nebyla, jen za cenu většího pohodlí při převodu řešení úlohy zpět na řešení hry.

7.3.3 Druhý způsob vytvoření úlohy

Základní myšlenka druhého způsobu je stále stejná. Povinná část úlohy musí zůstat bezezměny a nepovinnou část navrhneme úsporněji. Místo toho, abychom pro každou barvu na každém dílku měli jeden nepovinný sloupec, zakódujeme barvu binárním kódem. I nyní potřebujeme zajistit výběr správných řádků (barev sousedních dílků). Má-li mít dílek libovolnou barvu reprezentovanou jedinečným binárním kódem, pak sousednímu dílku přiřadíme inverzní kód. Tím si při zakrývání takových řádků vynutíme zachování nezakrytých sousedů požadované barvy ve struktuře.

Pro každou barvu tedy potřebujeme nejenom její binární kód, ale i její převrácený kód. Navíc každý tento kód musí být jedinečný. Potom počet bitů kódu (počet nepovinných sloupců pro jeden dílek) je roven $\lceil \log_2 (2 \cdot b) \rceil$, což je menší už pro počet barev $b = 4$.

Pro online verzi hry a tedy $b = 5$ (barvy A až E) je počet bitů roven $\lceil \log_2 (2 \cdot 5) \rceil = 4$, a kódy barev by mohly vypadat například takto:

Barva	Kód	Inverzní kód
A	0000	1111
B	0001	1110
C	0010	1101
D	0011	1100
E	0100	1011

Když si nyní budeme chtít vynutit na sousedovi například barvu B, tak bude mít hodnoty nepovinných sloupců pro konkrétní dílek rovny hodnotám 1110 (inverzní barva). Dojde tedy k zakrytí prvních tří sloupců (jedničky inverzního kódu), zakryjí se tedy barvy, které mají jedničku alespoň v jednom z prvních tří sloupců (jedničky kódu barvy). Zakryjí se tedy barvy C, D a E.

Požadovali jsme ale výběr barvy B. Když nyní budeme po algoritmu chtít aby pokryl všechny sloupce a to i ty nepovinné, potom výběr zbývající barvy A nepovede k řešení. Už by totiž nebyla žádná jiná možnost jak pokrýt poslední bit uvažovaného dílku. Jedinou možností, která vede ke správnému řešení je výběr požadované barvy B.

Tento druhý způsob vytvoření úlohy tedy nepracuje s modifikovanou verzí Dancing Links algoritmu. Je zapotřebí pokrýt všechny vytvořené sloupce!

7.3.4 Poznámky k druhému způsobu

Přednostně je důležité ještě jednou zdůraznit, že druhý způsob vytvoření úlohy nemá žádné nepovinné sloupce a vystačí si se základní verzí DLX algoritmu.

Tento způsob se zdá být sice paměťově méně náročný, což si ještě ověříme v následující kapitole, nedokáže ale ihned zajistit přesně požadovanou barvu sousedního dílku. Při rekurzi tak neredukuje úlohu stejně rychle jako první způsob. Navíc může na sousední dílky umístit různé barvy. S heuristickým výběrem sloupce snad ale svou chybu rozpozná dostatečně brzo.

Protože musíme pokrýt všechny sloupce, musíme také vytvořit řádek úlohy, který si vynutí okrajové rozmístění barev. Bez tohoto řádku, bychom nebyli schopni pokrýt všechny sloupce. Samotná implementace by ale mohla zajistit, aby byl kód okrajové barvy tvořen pouze jedničkami. Okrajový řádek by potom byl jen sadou nul a byl by tedy nepotřebný.

Převod řešení úlohy zpět na řešení hry je opět jednoduchý. Pokud bude označení sloupců například 6-1-1, což je podobné jako u prvního způsobu, tak už zbývá jen správně identifikovat barvu na dílku. K tomu nám bude sloužit právě poslední část označení, která určuje pořadové číslo bitu. Potom už není žádný problém poskládat kód barvy dílku a k tomuto kódu zjistíme jeho barvu pomocí stejného klíče, který barvám tyto kódy přiřazoval.

7.4 Výpočty

Pro číselné vyjádření obtížnosti hledání řešení hry Eternity II si spočítáme pár základních údajů pro určení velikosti úlohy. Určitě nás bude zajímat počet sloupců, řádků a elementů – jedniček v tabulce.

7.4.1 První způsob

Nechť n je velikost základny čtvercového hracího pole a b je počet barev včetně té okrajové, pak

- **počet sloupců** je roven $n^2 \cdot (1 + 4 \cdot b)$,

- **počet řádků** je nejvýše $4 \cdot n^4$,
- **počet elementů na řádku** je nejvýše $1 + 4 \cdot b$.

Počet sloupců je tvořen součtem povinných n^2 a nepovinných $n^2 \cdot 4 \cdot b$, kde hrací plocha má velikost n^2. Každé políčko má 4 dílky a každému dílku můžeme přiřadit jednu z b barev.

Počet řádků bude maximálně $4 \cdot n^4$ v závislosti způsobu vytváření úlohy a kontroly duplicitních řádků. Každou z n^2 kostek můžeme umístit na n^2 políček hrací plochy a 4x ji otočit.

Počet elementů na jednom řádku bude nejvýše $1+4 \cdot b$. Pro dosažení této hodnoty musí být kostička umístěna mimo okraj, aby si mohla vynucovat barvu okolních dílků dalších kostiček. Jedna jednička je pro označení pořadového čísla kostky (povinný sloupec), $4 \cdot (b-1)$ jedniček jsou jedničky označující barvu kostky na všech jejich čtyřech dílcích. A jednu jedničku pro každého souseda, u kterého si chceme vynutit barvu, nejvýše tedy 4 další jedničky.

Pro srovnání náročnosti jednotlivých verzí hry Eternity II nám poslouží následující tabulka.

	$n = 4, b = 5$ online verze	$n = 16, b = 5$	$n = 4, b = 23$	$n = 16, b = 23$ originální verze
počet sloupců	336	5 376	1 488	23 808
maximální počet řádků	1 024	262 144	1 024	262 144
max. elementů na řádku	21	21	93	93
max. elementů v úloze	21 504	5 505 024	95 232	24 379 392

7.4.2 Druhý způsob

- **počet sloupců** je roven $n^2 \cdot \Big(1 + 4 \cdot \lceil \log_2 (2 \cdot b) \rceil\Big)$,
- **počet řádků** je nejvýše $4 \cdot n^4$,
- **počet elementů na řádku** je nejvýše $1 + 4 \cdot \lceil \log_2 (2 \cdot b) \rceil$.

Počet sloupců je tvořen součtem sloupců pro pořadové číslo kostky (n^2) a sloupců pro každý dílek všech políček vynásobený počtem bitů pro kódování barev $n^2 \cdot 4 \cdot \lceil \log_2 (2 \cdot b) \rceil$.

Počet řádků bude maximálně $4 \cdot n^4$ stejně jako u prvního způsobu.

Počet elementů na jednom řádku je nejvýše $1+4 \cdot \lceil \log_2 (2 \cdot b) \rceil$. Rovnosti je dosaženo stejně jako u prvního způsoby pouze u dílků, které neleží na okraji hrací plochy.

Zde je opět tabulka výpočtů pro druhý způsob implementace hry Eternity II.

	$n = 4, b = 5$ online verze	$n = 16, b = 5$	$n = 4, b = 23$	$n = 16, b = 23$ originální verze
počet sloupců	272	4 352	400	6 400
maximální počet řádků	1 024	262 144	1 024	262 144
max. elementů na řádku	17	17	25	25
max. elementů v úloze	17 408	4 456 448	25 600	6 553 600

Druhým způsobem došlo k omezení velikosti úlohy, tak jak jsme očekávali. Čím více barev je v úloze použito, tím úspornější druhý způsob bude.

7.5 Popis vytvořených tříd

Pro úlohu Eternity II jsme vytvořili pouze funkční třídy pro sestavení úlohy pokrytí, neimplementovali jsme ale žádné uživatelské rozhranní.

7.5.1 Třída EternityTask

Tato třída uchovává informace o celé úloze. Jejími parametry jsou:

- **columns** – počet sloupců úlohy,
- **rows** – počet řádků úlohy,
- **List<EternityBrick> bricks** – seznam kostek,
- **List<Object> colours** – seznam barev,
- **borderColour** – barva okraje.

Navíc tato třída disponuje metodami, které umí zjistit index sousedního políčka:

- **getLeftNeighbourIndex(int onBoardIndex)**,
- **getUpNeighbourIndex(int onBoardIndex)**,
- **getDownNeighbourIndex(int onBoardIndex)**,
- **getRightNeighbourIndex(int onBoardIndex)**,

nebo spočítat počet bitů pro zakódování všech barev:

- **getBitsForColours()**.

7.5.2 Třída EternityTaskCreator a EternityTaskCreatorVersion2

Třída EternityTaskCreator rozšiřuje třídu TaskCreator (viz kapitola 6.1) o nové metody potřebné pro sestavení úlohy pokrytí Eternity II. Jejím hlavním rozdílem je, že pracuje se dvěma kořenovými objekty, kde jeden z nich představuje povinou část úlohy a druhý nepovinou.

Mezi její metody patří například:

- **createTaskTree(EternityTask task)** – vytvoření úlohy pokrytí,
- **addRow(HeaderAndDataObject mandatoryRoot, TreeSet<Integer> mandatoryPositions, HeaderAndDataObject optionalRoot, TreeSet<Integer>optionalPositions)** – přidání řádku do úlohy s rozlišením povinných a nepovinných sloupců,
- **addBorder(EternityTask task, HeaderAndDataObject mandatoryRoot, HeaderAndDataObject optionalRoot)** – přidání řádku, který si vynutí barvu na okraji hrací plochy,
- **interpretationOfSolution(Solution solution)** – velmi důležitá metoda, která převede nalezené řešení do čitelné formy.

Třída EternityTaskCreatorVersion2 přepisuje metody třídy EternityTaskCreator tak, aby byla úloha sestavená podle popisu v kapitole 7.3.3.

8 Závěr

Prvním cílem této diplomové práce bylo sestavit a naimplementovat algoritmy, které vytvoří úlohu rozkladu kompletního grafu na předepsané podgrafy jako úlohu pokrytí pro Dancing Links algoritmus, a následně budou umět tuto úlohu vyřešit.

Navíc jsme podle požadavků zadání tento algoritmus paralelizovali. Programovací jazyk C++ by byl pro hledání řešení jistě rychlejší, my jsme však zvolili Javu jako programovací jazyk a veškerá komunikace serveru s klientskými stanicemi probíhá přes TCP/IP, takže je možné serverovskou část umístit na počítač dostupný z internetu a bude se k němu moci připojit a podílet se na výpočtu řešení libovolný počet klientských stanic. Je tedy vše připraveno na řešení libovolně složitých úloh a přitom nás neomezuje nutnost spouštět program pouze na vícejádrových superpočítačích.

Pokud se tedy rozhodneme využít tohoto nástroje pro hledání rozkladů, tak by určitě bylo dobré vytvořit webové stránky věnující se tomuto problému s odkazy na stažení klientských částí, aby se do řešení problému mohl zapojit opravdu každý.

Je-li úkolem nalezení prvního řešení pro graf řádu $n = 16$ a dvojhvězdu jako jeho rozkládající podgraf, pak naimplementovaný algoritmus naleze toto řešení již během několika vteřin, což dokazuje efektivitu navrženého algoritmu. K ověření jeho správnosti jsme porovnali počty řádků, které nageneroval a které souhlasí s vypočítanými hodnotami. Stejně tak výsledná řešení určená programem jsou opravdu rozkladem kompletního grafu na zadané podgrafy.

Velkým přínosem naimplementovaných algoritmů je to, že dokaží již při generování úlohy odstranit identické řádky (izomorfní podgrafy). Pro úlohu se zadanými podmínkami izomorfismu, řádu $n = 16$ a dvojhvězdu jako podgraf, je nagenerovaná úloha 50 803 200x menší než, kdyby podmínky zadané nebyly.

Pro praktické použití by jistě bylo dobré navázat tak, aby podmínky symetrie nageneroval program sám bez pomoci uživatele.

Dalším cílem této práce je matematická reprezentace Eternity II skládačky. Vytvořené algoritmy dokáží demonstrační online verzi hry vyřešit během zlomku vteřiny. Při pokusu tuto úlohu zvětšit 4-krát na velikost hrací plochy 8x8 políček už ale nejsme ani během několika hodin schopni dojít k řešení. Tvůrce této skládačky, Christopher Monckton, si je zajisté dobře vědom toho, že vyřešit tuto úlohu výpočetní technikou nebude vůbec jednoduché a hlavně nijak rychlé a nejspíš právě proto stanovil výši výhry tak vysokou.

Pokud bychom chtěli ušetřit ještě další místo v operační paměti počítače, pak musíme vymyslet důmyslnější kódování barev úlohy, stejně tak by se určitě dala kódovat i pozice kostky na hrací ploše. To jsou jen jedny z mnoha teoretických návrhů, které stojí za zamyšlení.

Pro urychlení nalezení řešení by bylo dobré vymyslet způsob jak omezit počet nagenerovaných řádků, jak už jsme zmínili, například kontrolou jestli různá natočení kostky nevytvářejí identické řádky úlohy (symetricky zabarvená kostka jednou nebo dvěma barvami). Další kontrola by mohla ověřovat zda-li v úloze nejsou identické kostky.

Reference

[1] **Hitotumatu, Hirosi; Noshita, Kohei.** *A Technique for Implementing Backtrack Algorithms and its Application.* Elsevier North-Holland, Inc., 1979. ISSN:0020-0190.

[2] **Hliněný, Petr.** *Diskrétní matematika (456-533 DIM).* Verze 1.02. VŠB skripta, 2004-2006.

[3] **Knuth, Donald E.** *Dancing links.* c2000 [cit. 28.3.2010].
Dostupné z: <http://lanl.arxiv.org/PS_cache/cs/pdf/0011/0011047v1.pdf>.

[4] *Dancing links.* Poslední revize 12.1.2010 [cit. 28.3.2010].
Dostupné z: <http://en.wikipedia.org/wiki/Dancing_Links>.

[5] *Exact cover.* Poslední revize 13.12.2009 [cit. 28.3.2010].
Dostupné z: <http://en.wikipedia.org/wiki/Exact_cover>.

[6] *Knuth's Algorithm X.* Poslední revize 19.6.2009 [cit. 28.3.2010].
Dostupné z: <http://en.wikipedia.org/wiki/Algorithm_X>.

[7] *Eternity I.* Poslední revize 3.2.2010 [cit. 28.3.2010].
Dostupné z: <http://cs.wikipedia.org/wiki/Eternity_I>.

[8] *Eternity.* Poslední revize 22.6.2009 [cit. 28.3.2010].
Dostupné z: <http://www.mathpuzzle.com/eternity.html>.

[9] *Eternity II.* Poslední revize 20.1.2010 [cit. 28.3.2010].
Dostupné z: <http://cs.wikipedia.org/wiki/Eternity_II>.

[10] **Copyright © Eternity II – Tomy.** *Eternity II.* [cit. 28.3.2010].
Dostupné z: <http://cz.eternityii.com>.

[11] **M.R. Garey; D.S. Johnson.** *Computers and Intractability: a Guide to the Theory of NP-Completeness.* Vydavatel New York: W.H. Freeman, 1979-. ISBN 0-7167-1045-5.

Obsah přiloženého CD

- **obo024.pdf** – text diplomové práce,
- **DLX_program** – složka obsahující program popsaný v kapitole 6,
  - **DLX.jar** – jar soubor s programem,
  - **start_client.bat** – soubor pro spuštění klientské části programu,
  - **start_server.bat** – soubor pro spuštění serverové části programu,
- **SRC** – složka obsahující zdrojové kódy.