UNIVERZITA KOMENSKÉHO V BRATISLAVE

FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

Algoritmus X pri riešení drevených hlavolamov

Bakalárska práca

2021 **Erik Korbeľ**

UNIVERZITA KOMENSKÉHO V BRATISLAVE

FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

Algoritmus X pri riešení drevených hlavolamov

Bakalárska práca

Študijný program:	Aplikovaná informatika
Študijný odbor:	Informatika
Školiace pracovisko:	Katedra aplikovanej informatiky
Školiteľ:	RNDr. Peter Borovanský, PhD.

2021 **Erik Korbeľ**

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Erik Korbeľ
Študijný program: aplikovaná informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Algoritmus X pri riešení drevených hlavolamov
Solving wooden puzzles using Algorithm X

Anotácia: Cieľom práce je použiť Knuthov Algoritmus X primárne určený na riešenie problému Exact Cover Set problému pri hľadaní riešení netriválnych drevených hlavolamov a skladačiek, napr. z hry Ubongo. Cieľom práce je pomocou tohoto algoritmu navrhnúť mieru náročnosti pre človeka a následne navrhnúť aj veľmi ťažké zadania podobnej hry. Práca predpokladá pochopenie algoritmu X, testovanie na netriviálnych zadaniach a hľadanie limitov tohoto algoritmu pre existujúce drevené hlavolamy. Ďalším cieľom práce je vizualizovať algoritmom nájdené riešenie v nejakom grafickom prostredí.

Literatúra: Donald E.Knuth: The Art of Computer Programming, Volume 4, Fascile 5

Vedúci: RNDr. Peter Borovanský, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: prof. Ing. Igor Farkaš, Dr.

Dátum zadania: 12.08.2020

Dátum schválenia: 03.11.2020

doc. RNDr. Damas Gruska, PhD.
garant študijného programu

....
študent vedúci práce

Poďakovanie

Týmto by som sa chcel poďakovať všetkým ľudom, bez ktorých by som túto prácu nezvládol napísať. V prvom rade ďakujem RNDr. Petrovi Borovanskému, PhD., ktorého vedenie je jediný dôvod, prečo sa mi podarilo spraviť túto prácu takú kvalitnú, aká je. Ďalej ďakujem mojej rodine za to, že ma vždy podporovala. Najväčšia vďaka však patrí mojej životnej partnerke Lei Vrábelovej, ktorej existencia mi neustále dodávala silu dokončiť túto prácu, bez ohľadu na to, na aké veľké problémy som narazil.

Abstrakt

Moja bakalárska práca mala za cieľ riešiť drevené hlavolamy, pomocou relatívne neznámeho, ale efektívneho Algoritmu X, ktorý vytvoril Donald Knuth na riešenie problému úplného pokrytia, ktorý je ekvivalentný problému hľadania všetkých riešení hlavolamov. Konkrétne som sa rozhodol venovať hrám Genius Square a Ubongo 3D. Okrem riešenia týchto hlavolamov vie môj program aj určovať ich náročnosť a na základe toho je schopný hlavolamy aj generovať. Jednotlivé riešenia hlavolamov sa s použitím 3D renderovania aj dajú vykresliť. V tejto práci opisujem jej východiská, návrh riešení problémov zo zadania a taktiež ako bola práca implementovaná pomocou programovacieho jazyka Kotlin. Na koniec uvádzam aj výsledky testovania generátorov hlavolamov a mojej implementácie Algoritmu X.

Abstract

The main goal of my bachelor thesis was solving wooden puzzles using not well known, yet efficient Algorithm X, which has been created by Donald Knuth for solving exact cover problems. These problems are equivalent to finding all solutions of puzzles. The puzzles I chose to solve in this thesis are games called The Genius Square and Ubongo 3D. Besides solving these puzzles, my program can determinate their difficulty and using this function it is able to generate its own puzzles. Solutions of puzzles can be rendered in 3D using my program. My thesis contains descriptions of resources I used, design of solutions of problems and how I implemented things using programming language Kotlin. At last, I describe the results of testing of my puzzle generators and Algorithm X implementation.

Úvod 1

1. Východiská 2

1.1 NP-úplný problém 2

1.2 Spájaný zoznam 2

1.3 Problém úplného pokrytia 3

1.4 Algoritmus X 4

1.5 Technika “Dancing links“ 5

1.6 Problém úplného pokrytia pomocou 4-smerného cyklického spájaného zoznamu 5

1.7 Algoritmus X s použitím techniky “Dancing Links“ 7

1.8 Hlavolamy 12

1.9 Podobné práce 13

2. Návrh 14

2.1. Použité technológie 14

2.2. Riešenie problémov úplného pokrytia 14

2.3. Stavba hlavolamov 16

2.4. Redukcia hlavolamov 17

2.5. Náročnosť hlavolamov 18

2.6. Návrh hry Genius Square 19

2.7. Návrh hry Ubongo 3D 20

2.8. 3D Vykresľovanie hlavolamov 21

3. Implementácia 23

3.1. Implementácia Algoritmu X 23

3.2. Implementácia hlavolamov 25

3.3. Redukcia hlavolamov do problému úplného pokrytia 27

3.4. Určovanie náročnosti hlavolamov 29

3.5. Implementácia hry Genius Square 30

3.6. Implementácia hry Ubongo 3D 31

3.7. Vykresľovanie pomocou knižnice OpenGl 33

4. Testovanie 34

4.1. Štatistika všetkých riešení hry Genius Square 34

4.2. Štatistiky generovania nových kociek hry Genius Square 35

4.3. Štatistiky generovania nových zadaní hry Ubongo 3D 36

4.4. Benchmark pre Algoritmus X 36

Záver 38

Použitá literatúra 39

Úvod

Každý človek už niekedy riešil hlavolam, alebo hral nejakú hru, pri ktorej strávil niekedy až priveľa času rozmýšľaním. Existujú hlavolamy, ktoré sa zdá až nemožné vyriešiť. Presne takéto hlavolamy boli inšpiráciou pre zadanie mojej práce.

Prvou hrou, ktorú riešim je Genius Square. Je to jednoduchá hra, v ktorej hráč hodí hracími kockami, ktoré mu určia na aké pozície hracej plochy má umiestniť valčeky, ktoré sú ďalej nemenné. Následne sa hráč snaží umiestniť ďalších deväť jedinečných útvarov do hracej plochy tak, aby sa celá zaplnila. Ak sa to hráčovi podarí, hlavolam je úspešne vyriešený.

Druhá hra sa nazýva Ubongo 3D. V tejto hre má hráč tiež dostupné nejaké útvary, ktoré sa snaží navzájom poskladať tak, aby vytvoril stavbu výšky dva. To aký pôdorys má mať stavba, je určené zadaním hry. V tejto hre sú rôzne úrovne obťažnosti, teda existujú veľmi ľahké zadania hier, ale aj náročné.

Zrejme najdôležitejšia vec na celej práci je spôsob, akým sú hlavolamy riešené. Pri tom sa využíva ich transformácia na matematický problém zvaný úplne množinové pokrytie. Tieto problémy riešim pomocou netriviálneho Algoritmu X, ktorý vytvoril Donald Knuth presne na tento účel.

V prvej kapitole popisujem teoretické pozadie problému úplného pokrytia a fungovanie Algoritmu X a s tým súvisiacich algoritmov, z ktorých som vychádzal pri tvorbe programu. Taktiež sú tu uvedené všetky potrebné detaily o riešených hrách.

Druhá kapitola obsahuje návrh môjho riešenia, teda to akým spôsobom riešim problémy zo zadania a ako som si ich rozvrhol na rôzne podproblémy.

Nasledujúca kapitola už hovorí o konkrétnej implementácii v jazyku Kotlin. Ukážem ako som implementoval Algoritmus X, aké triedy som navrhol na reprezentáciu hlavolamov, ale aj akým spôsobom zrozumiteľne vykresľujem riešenia hlavolamov.

Posledná kapitola obsahuje štatistiky dĺžky behu niektorých častí môjho programu a výkonnostné porovnanie mojej implementácie Algoritmu X s inými už existujúcimi implementáciami.

1. Východiská

V tejto kapitole sa nachádzajú teoretické základy potrebné pre túto prácu. Začne s definíciami NP-úplného problému a spájaných zoznamov. Nasledovať bude vysvetlenie problému úplného pokrytia, Algoritmu X, techniky Dancing links a použitia techniky Dancing links pri Algoritme X. Ďalej budú vysvetlené pravidlá hier Ubongo 3D a Genius Square, ktorých zadania bude moja práca riešiť a nakoniec budú uvedené podobné práce.

1.1 NP-úplný problém

NP predstavuje skratku pre nedeterministický polynomiálny čas. Trieda NP problémov je charakteristická tým, že správnosť riešení týchto problémov je dokázateľná v polynomiálnom čase. NP-úplné problémy [4, s. 452] sú najnáročnejšie z pomedzi triedy NP, nie sú zatiaľ riešiteľné v polynomiálnom čase. Ide o jedny z najviac riešených problémov súčasnej informatiky. Ak o probléme je preukázané, že je NP-úplný, tak jeho riešenie bude mať minimálne exponenciálnu zložitosť. Klasickým príkladom takéhoto problému je Exact cover problém.

1.2 Spájaný zoznam

Jednosmerný spájaný zoznam

Ide o najjednoduchší spomedzi všetkých druhov spájaných zoznamov. Jednosmerný spájaný zoznam [5] je definovaný pomocou triedy **Node**, ktorá má atribúty *name* (názov) a smerník na nasledujúci objekt typu **Node**, zvaný *next*. Ak máme referenciu na prvý vrchol spájaného zoznamu, tak pomocou atribútu *next* ho vieme celý prejsť. Ak však nejaký vrchol má nejaký predchádzajúci vrchol, tak ku tomuto sa už dostať nedá, keďže referencie sú iba jednosmerné.

Dvojsmerný spájaný zoznam

Tento druh spájaného zoznamu je vylepšením jednosmerného. Definovaný je [6] pomocou triedy **Node**, ktorá na rozdiel od prechádzajúceho typu má ešte jeden smerník, nazvaný *prev*, odkazujúci na predchádzajúci vrchol spájaného zoznamu. Toto poskytuje veľkú výhodu,

pretože si stačí pamätať referenciu na ľubovoľný vrchol zoznamu a pomocou dvojsmerných smerníkov sa dá dostať ku všetkým ostatným vrcholom.

Cyklický spájaný zoznam

V prvých dvoch typoch spájaných zoznamov mal zoznam vždy začiatok a koniec. Cyklický spájaný zoznam [6] nemá ani jedno z toho. Dá sa spraviť z ľubovoľného necyklického spájaného zoznamu tým, že ako nasledovník posledného vrcholu sa nastaví prvý a ak ide o dvojsmerný zoznam, tak ako predchádzajúci vrchol prvého vrcholu sa nastaví posledný. Týmto vznikne cyklus. Takto sa aj v jednosmernom cyklickom spájanom zozname vieme dostať ku všetkým vrcholom, bez ohľadu na to, na ktorý máme referenciu.

1.3 Problém úplného pokrytia

Majme množinu prvkov S, úplné pokrytie (Exact cover) [1, s. 64] množiny S, je taká kolekcia S_1, S_2... S_n podmnožín množiny S, že zjednotenie množín S_1 až S_n tvorí množinu S a ich prienik je prázdna množina.

Problém úplného pokrytia množiny (Exact cover set problem) je problém toho, či existuje úplne pokrytie množiny.

$$A = \begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Obr. 1: binárna matica A, prevzaté z [1, s. 64].

Problém úplného pokrytia sa dá [1, s. 64] vyjadriť binárnou maticou. Na obrázku 1 možno vidieť binárnu maticu A, ktorá má sedem stĺpcov, ktoré pomenujeme A až G. Tieto tvoria množinu S, ktorá sa pokrýva. Má aj šesť riadkov, ktoré pomenujeme 1 až 6. Stĺpce matice v Algoritme X [kapitola 1.4] nazývame prvky (items), sú to prvky množiny S, ktoré treba pokryť a riadky matice nazývame možnosti (options). Každá možnosť je nejaká

podmnožina všetkých prvkov, podmnožina množiny S. Ak je v riadku jednotka, daná možnosť obsahuje daný prvok, ak tam je nula, neobsahuje ho.

Riešením problému úplného pokrytia [1, s. 64] je taká kolekcia disjunktných riadkov matice, ktorá bude mať v každom stĺpci presne jednu jednotku. Po krátkom preskúmaní je jasné, že jediným riešením problému zadaného v obrázku 1 je zjednotenie riadkov 1, 4 a 5, ktoré tvorí celú množinu S.

1 = {C, E}

4 = {A, D, F}

5 = {B, G}

1 ∪ 4 ∪ 5 = {A, B, C, D, E, F, G} = S

1.4 Algoritmus X

Je to algoritmus vytvorený Donaldom Knuthom na nájdenie všetkých riešení problému úplného pokrytia definovaného binárnou maticou A [2, s. 3]. Podstatou algoritmu je postupné skúšanie všetkých možností . Pseudokód algoritmu vyzerá nasledovne [2, s. 4]:

Ak A je prázdne, problém je vyriešený, program sa úspešne ukončí.
V opačnom prípade sa deterministicky vyberie stĺpec c.
Nedeterministicky sa vyberie riadok r, taký, že A[r, c] = 1.
R sa priradí do čiastočného riešenia.
Pre každé j také, že A[r, j] = 1:
    Zmaž stĺpec j z matice A.
    Pre každé i také, že A[i, j] = 1:
        Zmaž riadok i z matice A.
Opakuj rekurzívne algoritmus na redukovanej matici A.

Z pseudokódu je vidno, že algoritmus iba postupne vyberá stĺpce matice, vymaže ich a pre každý riadok matice, ktorý mal vo vymazanom stĺpci hodnotu 1, vymaže aj daný riadok z matice. Toto rekurzívne opakuje na zmenšenej matici, dokiaľ nedostane prázdnu maticu, čo indikuje, že bolo nájdené riešenie. Týmto najviac jednoduchým prevedením algoritmu sa síce dajú nájsť všetky riešenia problému, ale na veľkých vstupoch to je časovo neefektívne.

1.5 Technika "Dancing links"

Dancing links je technika vymyslená Donaldom Knuthom založená na jednoduchom princípe mazania a opätovného vracania vrcholov v dvojsmernom cyklickom spájanom zozname [1, s. 63]. Táto technika má veľké využitie pri backtrackingu, kde treba jednoducho odrobiť veľa zmien a veľmi efektívne zrýchľuje aj Algoritmus X.

Majme dvojsmerný cyklický spájaný zoznam, kde má každý vrchol X [1, s. 63] svoj predchádzajúci vrchol LLINK(X) a nasledujúci vrchol RLINK(X). Tieto sa dajú brať ako ekvivalenty atribútov *prev* a *next* odkazujúcich na objekty vrcholov spájaného zoznamu v triede **Node**, uvedených v kapitole 1.2. Vrchol X sa dá jednoducho vymazať zo zoznamu tak, že nastavíme [1, s. 63]:

RLINK(LLINK(X)) = RLINK(X)

LLINK(RLINK(X)) = LLINK(X)

Zvonku sa potom javí, že vrchol X v zozname nie je. Vnútorné smerníky prvku X ale stále ukazujú na pôvodný predchádzajúci a nasledujúci vrchol spájaného zoznamu. Tieto smerníky sú kľúčové pri opätovnom vracaní vrcholu X. Vrchol X sa dá vrátiť do zoznamu ďalšou jednoduchou dvojicou operácií [1, s. 63]:

RLINK(LLINK(X)) = X

LLINK(RLINK(X)) = X

Stačí si teda pamätať iba smerník na X, na to aby mohol byť X opätovne vrátený do zoznamu na miesto, kde pôvodne bol.

Pomocou tejto jednoduchej techniky je možné zo spájaného zoznamu pri rekurzívnom vnáraní mazať vrcholy a následne ich vracať na pôvodné miesto operáciou o zložitosti O(1). Na konci máme tú istú nezmenenú dátovú štruktúru, ktorá bola na začiatku.

1.6 Problém úplného pokrytia pomocou 4-smerného cyklického spájaného zoznamu

Na to, aby sme mohli použiť techniku Dancing links pri riešení problému úplného pokrytia pomocou Algoritmu X, je najprv potrebné, aby sme binárnu maticu prerobili do 4-smerného cyklického spájaného zoznamu [2, s. 5].

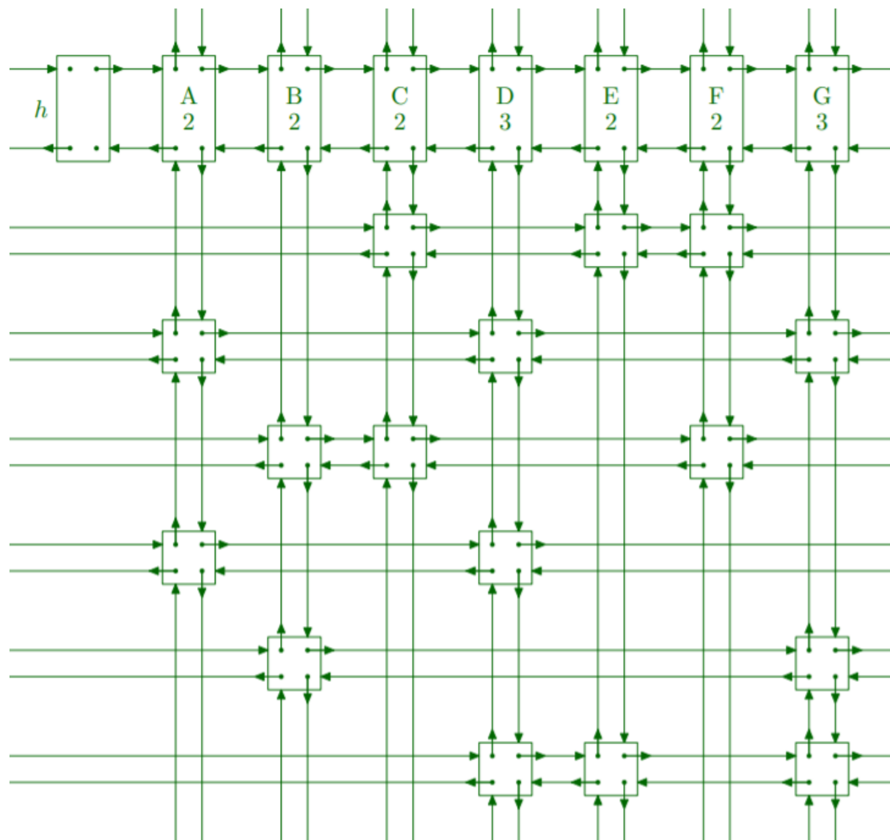
Obrázok 3 zobrazuje ako by sa problém úplného pokrytia zadaný maticou z obrázku 2 prerobil do takejto dátovej štruktúry. Je to vlastne štruktúra horizontálnych a vertikálnych cyklických spájaných zoznamov.

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Obr. 2: binárna matica, prevzaté z [2, s. 2].

Prvý horizontálny spájaný zoznam predstavuje prvky z množiny S, ktoré sa budú pokrývať. Prvý vrchol tohto zoznamu nazývame header (hlavička zoznamu), referencia na header bude zapamätaná v premennej. Každý vrchol X tohoto prvého horizontálneho spájaného zoznamu [2, s. 5] má RLINK(X), LLINK(X), ULINK(X), DLINK(X), LEN(X) a NAME(X), hlavička má len prvé dva atribúty nastavené. Tieto atribúty predstavujú ľavý, pravý, horný a dolný vrchol prislúchajúce ku vrcholu X. Posledné dve predstavujú počet možností, ktoré pripadajú ku danému prvku a názov prvku. Všetky vertikálne cyklické spájané zoznamy spájajú možnosti, ktoré obsahujú daný prvok. Zvyšné horizontálne cyklické spájané zoznamy predstavujú jednotlivé možnosti.

Riadky binárnej matice predstavujúce možnosti sa transformujú do horizontálnych cyklických spájaných zoznamov tak, že z jednotkových prvkov sa stanú vrcholy zoznamu a nulové sa ignorujú. Vrcholy predstavujúce možnosti [2, s. 5] majú atribúty RLINK(X), LLINK(X), ULINK(X), DLINK(X) a TOP(X). TOP predstavuje referenciu na vrchol z prvého horizontálneho zoznamu, predstavujúci prvok, ku ktorému vrchol z danej možnosti prilieha.



Obr. 3: 4-smerný cyklický spájaný zoznam, prevzaté z [2, s. 6].

1.7 Algoritmus X s použitím techniky "Dancing Links"

Majme problém úplného pokrytia transformovaný do 4-smerného cyklického spájaného zoznamu, ako bolo ukázane v kapitole 1.6. Aby sme mohli použiť Algoritmus X s pomocou techniky Dancing links, uvedieme si najprv pseudokód pomocných funkcií pracujúcich s touto dátovou štruktúrou [1, s. 66, 67]:

cover(i) (i predstavuje prvok - item) =

Nastav p = DLINK(i) (p, l a r sú lokálne premenné).

Dokiaľ p != i, hide(i), následne nastav p = DLINK(p) a opakuj.

Nastav l = LLINK(i), r = RLINK(i).

RLINK(l) = r, LLINK(r) = l.

hide(p) =

Nastav q = RLINK(p) a opakuj, kým q != p:

Nastav u = ULINK(q), d = DLINK(q), x = TOP(q).

Nastav DLINK(u) = d, ULINK(d) = u.

LEN(x) = LEN(x) – 1, q = RLINK(q).

Ďalej nasledujú 2 funkcie [1, s. 67] ktoré pomocou techniky Dancing links odrobia zmeny, ktoré prvé 2 funkcie vykonajú. Môžeme si všimnúť, že odrábajú zmeny presne v opačnom poradí ako boli spravené, pretože iba takto sa dátová štruktúra vráti do predchádzajúcej verzie. Pseudokód týchto funkcií vyzerá nasledovne:

uncover(i) =

Nastav l = LLINK(i), r = RLINK(i),

RLINK(l) = i, LLINK(r) = i.

Nastav p = ULINK(i).

Dokiaľ p != i, unhide(i), potom nastav p = ULINK(p) a opakuj.

unhide(p) =

Nastav q = LLINK(p) a opakuj, kým q != p:

Nastav u = ULINK(q), d = DLINK(q), x = TOP(q).

Nastav DLINK(u) = q, ULINK(d) = q.

LEN(x) = LEN(x) + 1, q = LLINK(q).

Samotné deterministické prevedenie Algoritmu X s použitím techniky dancing links by sa dalo pomocou pseudokódu napísať takto [1, s. 67]:

X1. [Inicializácia] Nastav problém v pamäti, tak ako na obrázku 3. Nastav $u = 0$.

X2. [Vstúp do úrovne u] Ak RLINK(header) = header (teda už boli pokryté všetky prvky), navštív riešenie špecifikované $x_0x_1x_2...x_{u-1}$ a choď do X8.

X3. [Vyber i] v tomto čase ešte musia byť prvky i_1, ..., i_t nepokryté, kde i_1 = RLINK(header), i_{j+1} = RLINK(i_j) a RLINK(i_t) = header. Vyber jeden z týchto prvkov a nazvi ho i.

X4. [Pokry i] Pokry prvok i pomocou funkcie cover(i) a nastav x_u = DLINK(i).

X5. [vyskúšaj x_u] Ak $x_u = i$, choď do X7 (vyskúšali sme všetky možnosti). V opačnom prípade nastav p = RLINK(x_u) a rob nasledovné dokiaľ p != x_u:

Nastav j = TOP(p).

Cover(j), nastav p = RLINK(p) (toto zakryje všetky prvky != i v možnosti, ktorá obsahuje x_u).

Nastav $u = u + 1$ a vráť sa do X2.

X6. [Vyskúšaj opäť] Nastav p = LLINK(x_u) a rob nasledujúce dokiaľ p != x_u:

Nastav j = TOP(p).

Uncover(j) a nastav p = LLINK(p). (Toto odokryje všetky prvky != i v možnosti, ktorá obsahuje x_u, pomocou reverznej operácii ku X5).

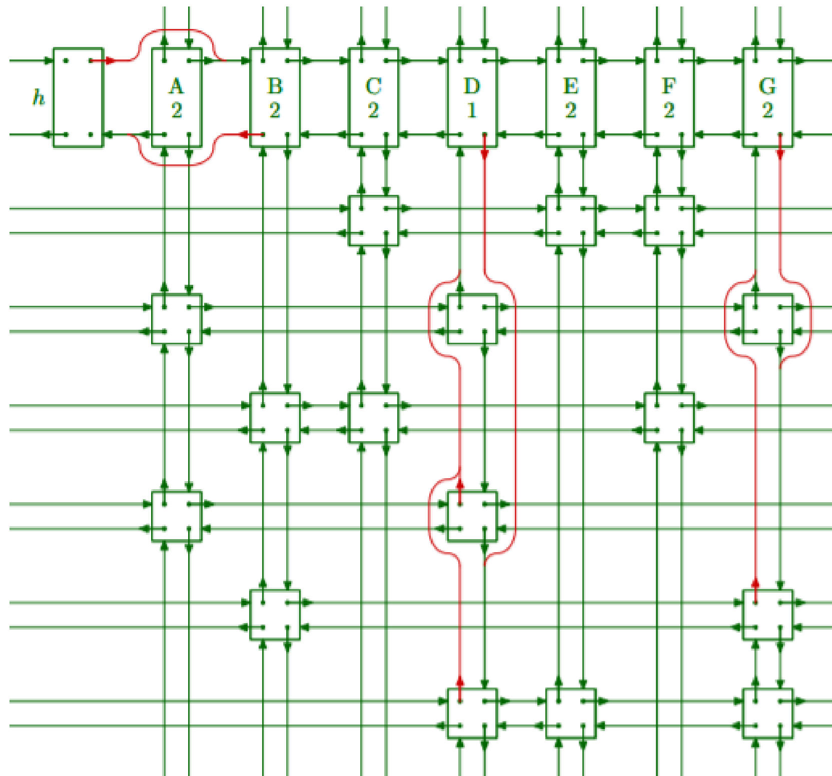
Nastav i = TOP(x_u), x_u = DLINK(x_u) a vráť sa do X5.

X7. [Backstrackuj] Odkry prvok i pomocou funkcie uncover(i).

X8. [Opusti úroveň u] Ukonči ak $u = 0$. V opačnom prípade nastav $u = u – 1$ a vráť sa do X6.

Algoritmus si môžeme predviesť na príklade z obrázku 3. Predpokladajme, že na úrovni 0 sa vyberie prvok A (na poradí výberu prvkov v konečnom dôsledku nezáleží, ale pre ilustráciu problému sme si zvolili prvok A) do premennej i. V kroku X4 sa zavolá cover(A). Dátová štruktúra po týchto krokoch bude vyzerať ako na obrázku 4.

Je vidno, že prvok A je pokrytý a rovnako aj všetky vrcholy, ktoré pripadali do jednotlivých možností obsahujúcich prvok A, okrem vrcholov pripadajúcich ku samotnému vrcholu A. Tieto sú ale pokryté tým, že vrchol A je pokrytý.



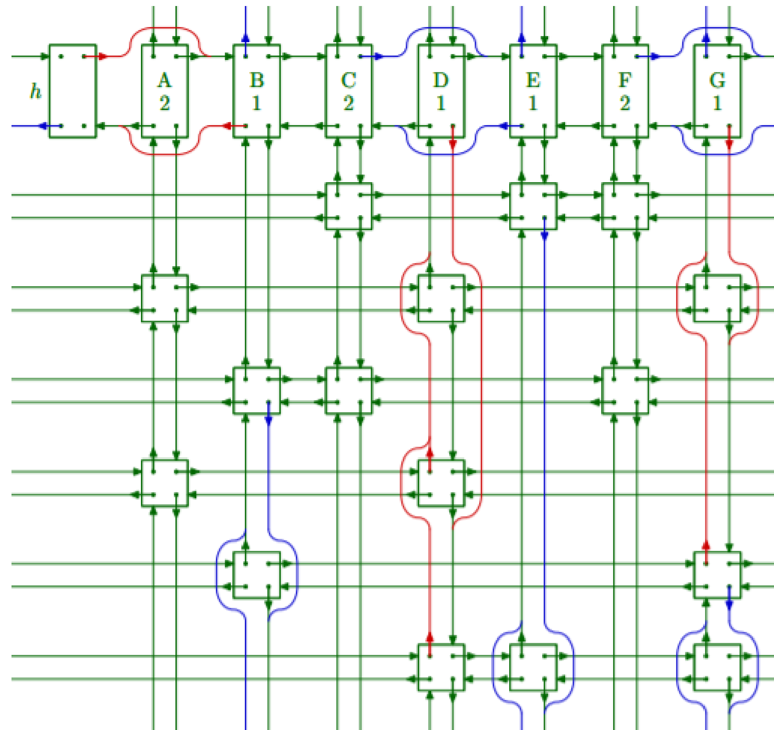
Obr. 4 Spájané zoznamy po pokrytí prvku A, prevzaté z [2, s. 7].

Do x_0 sa vyberie možnosť {A, D, G}. Po kroku X5 sa zakryjú aj prvky D a G a spájané zoznamy budú mať podobu z obrázku 5. Potom algoritmus vkročí do druhej úrovne.

V tej zostávajú už iba 2 možnosti: {C, E, F} a {B, C, F}. A keďže tieto možnosti nie sú disjunktné, nech si vyberie ktorýkoľvek prvok na pokrytie, nakoniec zistí, že v tejto vetve sa riešenie nenájde, keďže nebude možné pokryť všetky prvky.

Algoritmus sa pomocou X6, X7 a X8 vráti na úroveň 0, čím odrobí všetky doteraz spravené zmeny na dátovej štruktúre a do x_0 sa teraz vyberie možnosť {A, D}. V kroku X5 sa pokryje aj prvok D a ďalej sa vstúpi na úroveň 1. Zostanú 3 možnosti: {C, E, F}, {B, C, F} a {B, G}.

Na druhej úrovní budeme predpokladať, že sa algoritmus rozhodne pokryť prvok C (opäť nezáleží na tom, že to bude práve C, mohol by to byť aj ktorýkoľvek iný z ostávajúcich nepokrytých prvkov), vyberie teda jedinú možnosť {C, E, F} prislúchajúcu k prvku C a pokryje aj E a F. Ostane teda úž iba jediná nepokrytá možnosť {B, G} a iba 2 nepokryté prvky: B a G.



Obr. 5: Spájané zoznamy po pokrytí prvkov A, D a G, prevzaté z [2, s. 8].

Na tretej úrovni sa algoritmus pokúsi pokryť jeden z týchto dvoch prvkov a nezostane už žiaden nepokrytí prvok. Na začiatku ďalšej úrovne algoritmus zhodnotí, že bolo nájdené riešenie pozostávajúce z možností:

{A, D}

{C, E, F}

{B, G}

Vidíme, že všetko sú to disjunktné možnosti a ich zjednotenie je množina A až G, teda je to nepochybne riešenie. Toto je zároveň aj jediné riešenie tohto problému.

1.8 Hlavolamy

Hra Genius Square

Genius Square je 2D hlavolam s 62208 zadaniami. Hra obsahuje hraciu plochu o rozmeroch 6 x 6 štvorčekov. Stĺpce sú označené číslami 1 až 6 a riadky písmenami A až F. Ďalej obsahuje 7 hracích kociek, ktoré majú na každej strane nejakú pozíciu z hracej plochy a vyzerajú nasledovne:

1.kocka: F2,E1,F2,A5,A5,B6

2.kocka: A1,C1,D1,D2,E2,F3

3.kocka: F4,E5,F5,D5,E6,E4

4.kocka: D4,B4,E3,C4,C3,D3

5.kocka: A6,A6,F1,F1,F1,A6

6.kocka: C5,F6,A4,D6,C6,B5

7.kocka: B3,A3,C2,B2,A2,B1

Hra obsahuje aj 9 útvarov zložených z kvádrov rozmerov 1x1x1. Tieto útvary spolu s hracími kockami sú zobrazené na obrázku 6.

Obr. 6: Ukážka hry Genius Square,
prevzaté z https://i.ytimg.com/vi/toqZ0nm8Y5M/maxresdefault.jpg .

Samotný hlavolam funguje tak, že hráč hodí siedmimi kockami a na pozície, ktoré na kockách padnú umiestni na hraciu plochu valčeky. Toto sú zablokované pozície a ďalej

sa valčeky už nemôžu presúvať. Úlohou hráča je na hraciu plochu následne umiestniť aj 9 útvarov tak, že celá plocha bude zaplnená ako na obrázku 6. Pre každý z 62208 rôznych hodov kocky vznikne zadanie, ktoré má riešenie.

Hra Ubongo 3D

Ubongo 3D, ako samotný názov naznačuje je 3D hra. Jednotlivé zadania ako možno vidieť na obrázku 7 pozostávajú z kartičky, na ktorej je vyobrazený pôdorys stavby, ktorú treba postaviť vždy do výšky dva. Zadanie kartičky obsahuje aj zoznam 3D útvarov, zložených s kvádrov rozmerov 1x1x1, ktoré sa majú použiť na postavenie tohto tvaru.

Obr. 7: Ukážka hry Ubongo 3D,
Prevzaté z https://eshop.albi.sk/data/cache/thumb_750-750-12/products/36066/1597210842/35525_4.jpg?s=b47fa8eb6093daac68e2665bda67c8c7 .

1.9 Podobné práce

Práca Solving Sudoku efficiently with Dancing Links [3] hľadala riešenia hry sudoku pomocou Algoritmu X a techniky Dancing links. Vychádza z teoretických podkladov od Donalda Knutha a je postavená hlavne na zistení výhodnosti použitia Dancing links pri riešení problémov úplného pokrytia a hľadaní efektívnej redukcie Sudoku do problému úplného pokrytia. Taktiež motivuje softvérových vývojárov, aby hľadali už existujúce algoritmy na riešenie problémov, ako je v ich prípade Algoritmus X, namiesto riešení problémov pomocou hrubej sily.

2. Návrh

Táto kapitola, ako naznačuje už jej názov, obsahuje návrh celej práce. Pre každý dielčí problém, ktorý vychádza zo zadania, v samotnej podkapitole ukážem, ako som si ho rozvrhol.

Kapitola je rozdelená na osem podkapitol, počínajúc s použitými technológiami a následne postupujúcich od najnižšej logickej úrovne po najvyššiu: riešenie problémov úplného pokrytia, všeobecná štruktúra hlavolamov, ich prevod na problémy úplného pokrytia, určovanie náročnosti hlavolamov, následné samotné hry Genius Square a Ubongo 3D a nakoniec vykresľovanie riešení hlavolamov.

2.1. Použité technológie

Prácu som sa rozhodol implementovať v jazyku Kotlin [7] z dôvodu, že to je moderný, výkonný a perspektívny programovací jazyk. Program bol vyvíjaný vo vývojovom prostredí IntelliJ [8]. 3D interaktívne vykreslenie riešení hlavolamov som vyhotovil pomocou knižnice LWJGL [10], ktorá využíva veľmi populárnu knižnicu OpenGL [9] na renderovanie 3D objektov.

2.2. Riešenie problémov úplného pokrytia

Najviac kľúčovým problémom bol samotný Algoritmus X, schopný riešiť ľubovoľný problém úplného pokrytia. Tento je nevyhnutný pri riešení hlavolamov, a preto bolo aj dôležité si ho dobre navrhnúť, keďže zvykne platiť, že neskoré úpravy v najnižších logických úrovniach zvyknú byť náročné.

Prvky množiny

V kapitole 1.3 bolo vysvetlené, že pri probléme úplného pokrytia sa pokrýva množina S. Touto množinou môžu byť prirodzené čísla, ale v mojej práci to častokrát búdu iné množiny, konkrétne množiny útvarov hlavolamu a základných prvkov hlavolamu, ktoré bude treba pokryť na to, aby sa hlavolam vyriešil. Kvôli univerzálnosti návrhu sa v ňom teda nebude hovoriť o konkrétnych množinách jedného určitého typu, ale jednoducho o množinách ľubovoľných prvkov.

Binárna matica

Problémy úplného pokrytia som sa rozhodol reprezentovať binárnou maticou, tak ako to bolo spomenuté aj v kapitole 1.3, pričom ich význam je tiež rovnaký. Stĺpce matice predstavujú jednotlivé prvky množiny, ktorej úplne pokrytie sa hľadá. Riadky matice predstavujú možnosti čiastočného pokrytia množiny - nejaké podmnožiny danej množiny. Úplné pokrytie množiny následne tvoria riadky matice, pre ktoré platí, že dokopy majú v každom stĺpci presne jednu jednotku, teda že sú schopné dokopy pokryť každý prvok množiny presne raz.

Algoritmus X

Algoritmus X dostane ako vstup binárnu maticu, ale nepoužíva priamo túto maticu, ale štvorsmerne cyklicky spájaný zoznam, vytvorený z matice, ako bolo opísané v kapitole 1.6. Algoritmus si teda najprv prevedie maticu na spájaný zoznam a potom z neho hľadá všetky úplné pokrytia. Podrobnejší opis fungovania algoritmu sa nachádza v kapitole 3.1.

S heuristika

Algoritmus X počas svojho behu viackrát vyberá prvok, ktorý bude pokrývať. To akým spôsobom sa bude prvok na pokrytie vyberať vie veľmi ovplyvniť to, ako rýchlo algoritmus nájde všetky riešenia problému.

V jednoduchšej verzii by Algoritmus X [2, s. 6] vybral prvok, ktorý treba pokryť tak, že si zvolí prvý prvok spájaného zoznamu a ten pokryje. Toto riešenie funguje dobre, pretože treba pokryť vždy všetky prvky, a je aj veľmi jednoduché na naprogramovanie, keďže sa vždy berie prvý prvok napravo od hlavičky spájaného zoznamu. Toto riešenie je síce jednoduché, ale pri veľkých problémoch sa ukazuje ako nevýhodné.

Knuth navrhuje [2, s. 4] použitie inej technológie výberu prvku na pokrytie. Konkrétne spomína, že algoritmus si vyberie z nepokrytých prvkov vždy prvok, ktorý sa vyskytuje v najmenšom počte možností. Tento spôsob výberu nazýva S heuristika a podľa [2, s. 9, 10] zrýchľuje beh algoritmu X. V mojej práci sa používa presne táto heuristika na výber prvku na pokrytie. Jej použitie vedie k menšiemu počtu rekurzívnych vnáraní, teda aj ku rýchlejšiemu nájdeniu všetkých riešení.

2.3. Stavba hlavolamov

Medzi problémom úplného pokrytia a hrami, ktoré táto práca rieši sa nachádza ešte jedna logická úroveň. Tou je úroveň popisujúca všeobecnú stavbu hlavolamov. Táto úroveň obsahuje všetky abstrakcie umožňujúce riešiť široké spektrum 2D a 3D hlavolamov pomocou Algoritmu X, vrátane hier Ubongo 3D a Genius Square. V tejto podkapitole sa nachádza popis návrhu všetkého, čo patrí na túto logickú úroveň: stavba hlavolamov, zadania hlavolamov, redukcia hlavolamov na problémy úplného pokrytia a spôsoby určovania náročnosti hlavolamov.

Všeobecný základný bod

Každý hlavolam má nejakú základnú stavebnú jednotku, z ktorej je zložená každá štruktúra, ktorú obsahuje. V prípade 2D hlavolamov to je väčšinou štvorec o rozmeroch 1x1 (zobrazené na obrázku 8) a v prípade 3D hlavolamov to je kocka o rozmeroch 1x1x1 (zobrazené na obrázku 8).

Obr. 8: základný štvorec a základná kocka.

Toto nie je nutnosť, hlavolamy môžu byť zložené aj z guličiek, trojuholníkov alebo rôznych nepravidelných útvarov, ale pre hlavolamy, ktoré rieši moja práca to platí, a preto budem ďalej považovať toto za základne stavebné jednotky.

Miestnosť hlavolamu

Miestnosť hlavolamu, alebo inak povedané priestor, ktorý sa v hlavolame vypĺňa, je množina základných stavebných jednotiek hlavolamu.

Útvar

Podstatou hlavolamov Genius Square, Ubongo 3D a aj iných populárnych hier je umiestňovanie rôznych útvarov tak, aby vyplnili priestor hlavolamu. Útvary sú množiny

základných stavebných jednotiek hlavolamu podobne ako priestor hlavolamu. Príklad 2D útvaru, aký sa vyskytuje v hre Genius Square je vidno na obrázku 9 naľavo a príklad 3D útvaru z hry Ubongo 3D napravo na tom istom obrázku. 2D tvary sú zložené z štvorčekov a 3D zase z kociek.

Obr. 9: 2D útvar a 3D útvar.

Taktiež musí platiť, že každý útvar je podmnožina priestoru hlavolamu, inak útvar nie je možné do hlavolamu vložiť.

Zadanie hlavolamu

Hlavolam pomocou opísaných štruktúr vie byť zadaný dvoma vecami - priestorom hlavolamu a množinou útvarov. Ak by zadanie hlavolamu obsahovalo útvary, ktoré nie sú podmnožiny priestoru hlavolamu, takýto hlavolam by nemal riešenie.

2.4. Redukcia hlavolamov

Ako sa uvádza v tejto kapitole, na priestor hlavolamu a útvary, ktorými sa hlavolamy vypĺňajú, sa dá pozerať ako na množinu základných stavebných jednotiek. Taktiež na každý z útvarov, ktorými sa priestor hlavolamu vypĺňa, sa dá pozerať ako na jeho podmnožinu. Riešením hlavolamov je množina útvarov, pre ktoré platí, že dokopy presne pokrývajú priestor hlavolamu, teda riešením hlavolamu je disjunktná množina útvarov, ktorej zjednotením je presne celý priestor hlavolamu.

Ako sa však transformuje hlavolam zadaný týmto spôsobom na problém úplného pokrytia, aby ho mohol riešiť algoritmus X?

Tento proces je jednoduchý, najprv si treba uvedomiť čo Algoritmus X potrebuje. Algoritmus pracuje s množinou prvkov, ktorých úplne pokrytie sa hľadá a maticou, ktorá reprezentuje všetky možnosti čiastočného pokrytia množiny.

Prvky množiny, ktorá sa pokrýva je množina bodov miestnosti hlavolamu, pretože treba presne pokryť každý bod hlavolamu.

Vytvorenie matice je viacfázový proces. Majme dostupné tvary, ktorými sa bude hlavolam pokrývať, všetky možnosti pokrytia prvkov [1, s. 250, 251] tvarmi sa získajú nasledovne:

- Pre každý dielik zostrojíme všetky jeho rotácie, pričom v závislosti od tvaru špecifického útvaru, môžu byť niektoré z týchto rotácií totožné, teda celkový počet unikátnych rotácií môže byť pre rôzne útvary odlišný.
- Pre každú unikátnu rotáciu každého útvaru vyskúšame akými všetkými spôsobmi sa dá umiestniť do hlavolamu. Toto sa dosiahne posunom útvaru po bodoch hlavolamu. Každé takéto posunutie rotácie útvaru po hlavolame, kde žiadna časť útvaru nepretŕča z hlavolamu v žiadnej osi je považovaná za možnosť pokrytia hlavolamu.

Pre jednotlivé možnosti pokrytia hlavolamu je už ich transformácia do riadkov matice jednoduchý proces. V riadku matice tvoria jednotlivé stĺpce prvky priestoru hlavolamu. Ak dané posunutie rotácie nejakého útvaru leží na danom dieliku, v stĺpci riadku bude 1, v opačnom prípade tam bude 0.

2.5. Náročnosť hlavolamov

Prvotnou myšlienkou toho ako určiť náročnosť hlavolamu, bolo zameriavať sa čisto na počet riešení hlavolamu. Podstatou tejto myšlienky bolo, že čím viacej riešení hlavolam má, tým ľahší musí byť, keďže je väčšia šanca, že človek na jedno z tých riešení narazí nejak náhodou, naopak ak má hlavolam len jedno riešenie, tak musí byť ťažké nájsť zrovna to jedno jediné riešenie. Tento spôsob však nie je ani zďaleka presný. V prípade veľmi jednoduchého hlavolamu, ktorý by obsahoval iba dva tvary a mal by jedno riešenie, by táto metodika tvrdila, že to je veľmi náročný hlavolam, napriek tomu, že pre každého človeka je triviálny.

Aby sa predišlo takýmto nepresným meraniam náročnosti, používam v mojej práci na meranie náročnosti mierne náročnejší algoritmus. Vychádza z rátania presnej pravdepodobnosti toho, že ak sa útvary náhodne vkladajú do hlavolamu, aká je šanca, že sa umiestnia tak, že dajú nejaké z riešení hlavolamu.

Prvým krokom tohto výpočtu je nájsť všetky riešenia hlavolamu, následne z daných riešení pre každý útvar zistiť koľkými rôznymi spôsobmi je v týchto riešeniach umiestnený.

Ďalej sa pre každý tvar zistí koľko je celkových možných umiestnení útvaru do hlavolamu.

Nakoniec sa pre každý tvar vypočíta podiel toho koľko rôznych umiestnení v riešeniach hlavolamov má a toho, koľko je všetkých možných umiestnení. Tieto podiely prislúchajúce ku všetkým útvarom sa navzájom vynásobia a vznikne výsledne číslo z rozsahu od 0 do 1 určujúce pravdepodobnosť toho, že náhodným ukladaním útvarov sa dôjde ku riešeniu hlavolamu. Toto číslo vyjadruje náročnosť hlavolamu.

Ak má hlavolam náročnosť 0, znamená to, že nemá žiadne riešenie, ak má náročnosť 1, má riešenie úplne vždy. Vo všeobecnosti, čím bližšie ku 0 je hodnota náročnosti, tým náročnejší je. Toto umožňuje porovnávať hlavolamy z hľadiska obťažnosti. Taktiež to napomáha vytvárať veľmi jednoduché, ale aj extrémne náročne zadania hlavolamov, ktoré dajú človeku zabrať.

2.6. Návrh hry Genius Square

Hra pozostáva z viacerých častí:

- Hracia plocha o rozmeroch 6x6.
- Deväť rôznych a jedinečných útvarov.
- Sedem totožných valčekov.
- Sedem kociek s názvami políčok hracej plochy.

Pomocou týchto súčastí vznikne zadanie hry tak, že hráč hodí všetkými kockami a hodnoty, ktoré na nich padnú, určia kam umiestni valčeky. Následne hráč nájde riešenie tak, že umiestňuje deväť jedinečných útvarov, aby presne vyplnil zvyšok hracej plochy.

Formalizácia zadania tejto hry na všeobecné zadanie hlavolamu ako bolo opísane v kapitole 2.3 je takáto:

- Priestor hlavolamu, ktorý sa bude pokrývať tvorí 36 štvorčekov hracej plochy.
- Útvary, ktorými sa priestor hlavolamu pokrýva, je sedem valčekov (ktoré sa dajú v postate predstaviť ako štvorčeky o rozmeroch 1x1 - teda základné jednotky hlavolamu) a taktiež aj deväť jedinečných a navzájom rôznych útvarov, ktoré sú súčasťou hry.

Toto zadanie sa pretransformuje na problém úplného pokrytia rovnako ako v kapitole 2.4, s jedným rozdielom - pre valčeky sa nebudú hľadať všetky možné uloženia v hlavolame, keďže ich poloha je predom predpísaná.

2.7. Návrh hry Ubongo 3D

Redukcia hry do problému úplného pokrytia

Pre zadania tejto hry platia viaceré pravidlá. Priestor hlavolamu je vždy stavba vysoká dve základne jednotky. Pôdorys priestoru však môže byť rôzny a teda aj celkový počet základných stavebných jednotiek, kociek s rozmerom jedna, ktoré zadanie hry tvoria, môže byť rôzny. Čo sa týka útvarov, z ktorých sa stavba má skladať, tie môžu byť pri každom zadaní rôzne, keďže aj pôdorysy hlavolamov sú rôzne. Vždy ale platí, že v zadaní sa každý tvar vyskytuje iba raz, nikdy sa neopakuje.

Transformácia na všeobecné zadanie hlavolamu popísane v kapitole 2.3 je priamočiara - všetky kocky, z ktorých sa skladá priestor hry sú miestnosť hlavolamu, ktorá sa bude vypĺňať a útvary, ktorými sa bude pokrývať, je množina 3D tvarov, ktoré určí zadanie hry. Transformácia tohto zadania na problém úplného pokrytia je presne rovnaká ako bolo opísané v kapitole 2.4.

Generovanie nových hier

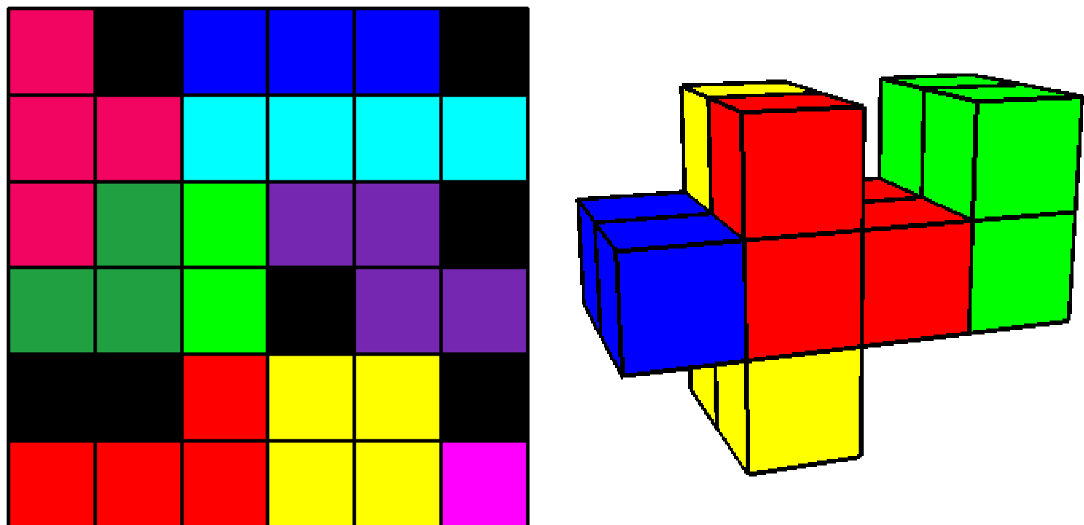
Program na generovanie vlastných zadaní hier Ubongo 3D je viacfázový proces. Jeho podstatou je vytvoriť hlavolam, ktorý má určitú požadovanú náročnosť. Algoritmus generovania začne s prázdnou miestnosťou hlavolamu a nejakou veľkou konštantnou množinou 3D útvarov a cyklicky opakuje tieto činnosti:

- Zistí či miestnosť hlavolamu nie je priveľká, ak je, vyprázdni ju a miestnosť sa generuje od začiatku.
- Vygeneruje dvojicu čísel x, y, takú že obe čísla sú z rozsahu od 0 do 5 a platí, že bod zo súradnicami [x, y, 0] sa ešte nenachádza v množine bodov, ktoré tvoria priestor hlavolamu, a zároveň susedí s nejakým bodom, ktorý sa tam nachádza. Do množiny bodov tvoriacich priestor hlavolamu následne pridá body so súradnicami [x, y, 0] a [x, y, 1] (z dôvodu, že zadania Ubongo 3D sú vždy vysoké dva dieliky).

- Skúsi vyriešiť hlavolam, zadaný priestorom, ktorý si buduje a veľkou konštantnou množinou útvarov.
- Ak daná miestnosť má nejaké riešenie, kde sa každý tvar, ktorý obsahuje nachádza iba jeden krát, zapamätá si množinu útvarov, ktoré sa v tomto riešení nachádzajú (táto množina je nejaká podmnožina konštantného zoznamu útvarov v pamäti).
- Zistí náročnosť hlavolamu tvoreného týmto priestorom a množinou útvarov. Ak má hlavolam takú náročnosť, aká bola požadovaná, cyklus skončí, v opačnom prípade pokračuje ďalším opakovaním cyklu.

Po skončení cyklu je vygenerované zadanie hry Ubongo 3D s požadovanou náročnosťou a môže byť ďalej použité na nájdenie všetkých riešení a následné vykreslenie riešení.

2.8. 3D Vykresľovanie hlavolamov



Obr. 10: Riešenie hry Genius Square a hry Ubongo 3D zobrazené pomocou 3D vykresľovania.

V prípade 3D vykresľovania hlavolamov nie sú žiadne vážnejšie problémy. Jeho pointou je vykresliť riešenia hlavolamov v trojrozmernom priestore. Pri tomto platí, že jednotlivé útvary, tvoriace zložený hlavolam sú jednoznačne farebne rozlíšené a tiež je jasne viditeľné okraje jednotlivých stavených jednotiek, z ktorých sú útvary zložené. Obrázok 10 ukazuje ako sa pomocou 3D dajú zobraziť riešenia hlavolamov. Táto technológia je síce vhodnejšia

na 3D hry ako je Ubongo 3D, ale použiteľná je aj na 2D hry ako Genius Square, ktoré nie je žiaden problém zobraziť v 3D priestore. Na to aby 3D zobrazenie dalo človeku čo najviac informácií, s riešeniami hlavolamov sa môže rôzne manipulovať, napríklad otáčať ich, približovať ich, alebo v prípade veľkých 3D hlavolamov aj rozložiť si ich, útvar po útvare, a pozrieť sa ako sú poskladané vo vnútri.

3. Implementácia

Táto kapitola popisuje algoritmické problémy, ktorých implementácia bola najväčšia výzva, ale aj také, ktoré sú subjektívne najviac zaujímavé. Ukážem fungovanie Algoritmu X a prevod hlavolamov na problém úplného pokrytia. Taktiež ako som odstránil symetrické riešenia hlavolamov a spôsob rátania náročnosti hlavolamov. Potom ako sú implementované a generované zadania hier Genius Square a Ubongo 3D a nakoniec zopár detailov ohľadom 3D renderovania.

3.1. Implementácia Algoritmu X

Modul obsahujúci Algoritmus X a všetky pomocné štruktúry, sa volá **algorithmX**. Je nezávislý od zvyšku môjho programu, na úrovni jeho abstrakcie žiadne hlavolamy neexistujú, iba problémy úplného pokrytia.

Výstup s riešeniami

Výstupom Algoritmu X sú riešenia problému úplného pokrytia. Interface **Printer** predpisuje všetko potrebné na prácu s riešeniami.

```
interface Printer {

    val solutions : MutableList<List<List<Name>>>

    fun reset() {
        solutions.clear()
    }

    fun add(solution : List<List<Name>>) {
        solutions.add(solution)
    }

    fun print()

}
```

Atribút *solutions*, je určený na riešenia problému a metódy majú takúto funkcionalitu:

- Metóda *reset* slúžiaca na vymazanie všetkých riešení z atribútu *solutions*, aby inštancia mohla byť použitá opätovne.
- Metóda *add* na pridanie nového riešenia do zoznamu riešení *solutions*.

- Metóda *print*, ktorá má za účel vypísať riešenia zo zoznamu *solutions*, podľa princípu, aký si zvolia triedy, ktoré tento interface implementujú.

Interface teda nepredpisuje špecifický spôsob alebo formát vypísania riešení, iba hovorí ako narábať s riešeniami tak, aby to bolo kompatibilné s mojou implementáciou Algoritmu X.

Algoritmus X

Moja implementácia algoritmu vychádzala priamo z pseudokódu od Knutha z kapitoly 1.7. Konštruktor triedy je takýto:

```
class AlgorithmX(matrix : Matrix,
                 names : List<Name>,
                 private val printer : Printer,
                 private val allSolutions : Boolean)
```

Význam parametrov konštruktora tejto triedy je:

- Binárna matica problému úplného pokrytia**.**
- Zoznam mien prvkov množiny, ktorej úplne pokrytie sa hľadá.
- Objekt triedy implementujúcej interface **Printer**, pomocou ktorého sa vypíšu riešenia.
- Boolean určujúci, či má algoritmus nájsť iba jedno riešenie problému, alebo všetky.

Pôvodný kód Knutha potreboval zopár úprav pred tým, než sa dal použiť. Jeho kód vychádzal zo starých programovacích techník a pre jazyk Kotlin bol viac-menej nepoužiteľný. Po prepísaní pomocou rekurzie vyzerá takto:

```
fun solve() {
    if (header.rLink == header) {
        printer.add(createResult())
    } else {
        val item = selectItem()
        var x = item.dLink
        cover(item)
        while (x != item) {
            visited.add(x)
            rowCover(x)
            solve()
            rowUnCover(x)
            visited.remove(x)
            x = x.dLink
        }
        unCover(item)
    }
}
```

Kód najprv otestuje, či v spájanom zozname sú ešte vrcholy, predstavujúce prvky pokrývanej množiny, a ak nie, pridá riešenie do objektu **Printer**. V opačnom prípade zvolí prvok na pokrytie, pokryje ho a následne rekurzívne skúša na menšom spájanom zozname. Výber prvku na pokrytie prebieha pomocou S-heuristiky opísanej v kapitole 2.2.

3.2. Implementácia hlavolamov

Táto podkapitola popisuje modul **cubes**, ktorý obsahuje implementáciu útvarov, priestorov hlavolamov a aj samotné zadania hlavolamov.

Všeobecný základný bod

Univerzálnosť tejto implementácie spočíva vo vytvorení interface **Point**, ktorý predpisuje všetky metódy, ktoré potrebujú všetky triedy, ktoré priamo narábajú s bodmi hlavolamov, bez toho aby obsahoval konkrétnu implementáciu týchto problémov, teda interface neobsahuje žiadne informácie o štvorcoch ani kockách. Medzi funkcionalitu metód patrí napríklad vytvorenie kópie bodu, posunutie bodu alebo rotácia bodu pomocou násobenia matíc, ako je zobrazene nižšie:

```
interface Point {
    ...
    fun copy() : Point
    fun set(other: Point) : Point
    fun add(other : Point) : Point
    fun subtract(other : Point) : Point
    fun rotate(matrix: Matrix) : Point
    ...
}
```

Vytvoril som aj jednu triedu implementujúcu tento interface, ktorú som nazval **Point3D**, a ktorá hovorí už konkrétne o trojdimenzionálnom bode. **Point3D** má 3 atribúty, ktoré sa nastavujú priamo v konštruktore:

```
class Point3D(var x : Int, var y : Int, var z : Int) : Point
```

X, *y* a *z* sú súradnice bodu. Keďže základná stavebná jednotka 3D hlavolamov, ktoré táto trieda rieši je kocka 1x1x1, tak atribúty *x*, *y* a *z* hovoria, kde má kocka počiatočné súradnice a jej všetky rozmery sú vždy 1, teda na popis kocky stačí jedna inštancia triedy **Point3D**. Pomocou nej sa dajú opísať aj štvorce, akurát ich *z* súradnica bude vždy rovná 0. Trieda **Point3D** teda slúži ako základná štruktúra pre všetky hlavolamy mojej práce, či už 2D alebo 3D.

Priestor hlavolamu

Priestor hlavolamu reprezentuje trieda **Puzzle**:

```
class Puzzle(val points : Set<Point>)
```

Jediný parameter konštruktora predstavuje množinu bodov hlavolamu - množinu inštancií tried implementujúcich interface **Point**.

Útvar

Tvary sú množina základných stavebných jednotiek, v mojej implementácii množina inštancií tried implementujúcich interface **Point**. Ako už bolo napísané aj v tejto kapitole, základné jednotky 2D aj 3D útvarov reprezentujem pomocou triedy **Point3D**, ktorá tento interface implementuje, pričom 2D útvary sa od 3D líšia iba v tom, že ich *z* súradnica je vždy rovná 0. Trieda **Shape** reprezentuje útvary.

```
class Shape(val name : String, points : Set<Point>)
```

Na konštruktore triedy je vidno, že obsahuje množinu bodov (atribút *points*) a taktiež aj názov tvaru (atribút *name*).

Zadanie hlavolamu

Na reprezentáciu zadania hlavolamu u mňa slúži trieda **Task**:

```
class Task(val puzzle: Puzzle,
           val shapes : Set<Shape>,
           val repeatableShapes : Set<Shape>,
           val fixedShapes : Set<Shape>)
```

Význam atribútov je:

- Atribút *puzzle* - objekt triedy **Puzzle** popisujúci miestnosť hlavolamu.
- Atribút *shapes* - množina tvarov, ktoré sa v hlavolame musia umiestniť presne raz.
- Atribút *repeatableShapes* - množina tvarov, ktoré sa v hlavolame môžu vyskytovať ľubovoľný počet krát.
- Atribút *fixedShapes* - množina tvarov, ktoré sa v hlavolame musia nachádzať na presne určenej pozícii.

Atribút *puzzle* je jediný povinne rôzny od Null hodnoty, ostatné môžu byť hocijak nakombinované, čim umožňujú použitie triedy **Task** na veľa typov zadaní hlavolamov.

3.3. Redukcia hlavolamov do problému úplného pokrytia

Riešenie hlavolamov

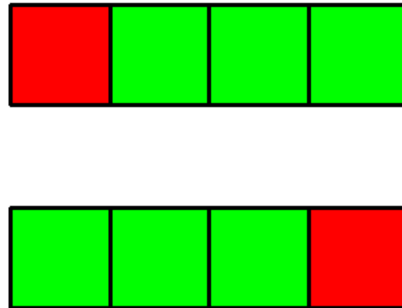
Modul **cubes** disponuje aj ďalšou esenciálnou funkcionalitou, ktorá je obsiahnutá v triede **Formalisation**. V triede sa nachádza metóda *createMatrixAndNamesFromTask*, ktorá dostane na vstupe zadanie hlavolamu ako inštanciu triedy **Task** a jej výstupom je dvojica - matica popisujúca problém úplného pokrytia, vytvorená zo zadania hlavolamu, a zoznam prvkov množiny, ktorá sa bude pokrývať. Tento výstup je priamo použiteľný ako vstup pre moju implementáciu algoritmu X. Takto sa zo zadania hlavolamu dajú získať všetky riešenia hlavolamu. Moja implementácia tohto postupu je nasledovná:

```
private fun fillMatrixWithShapes(shapes : Set<Shape>) {
    for (shape in shapes) {
        for (rotationMatrix in rotationMatrices) {
            shape.rotate(rotationMatrix).normalise()
            for (point in task.puzzle.points) {
                shape.translate(point)
                if (isShapeGoodToBeAdded(shape)) {
                    addToRow(shape)
                }
                addSymmetries(shape)
            }
        }
    }
}
```

Každý útvar sa postupne otáča, posúva, kontroluje a nakoniec pomocou metódy *addToRow* transformuje na riadok matice.

Odstránenie symetrických riešení

V metóde *fillMatrixWithShapes* sa objavuje aj volanie metódy *addSymmetries*. Pomocou nej sa zabezpečuje, aby v matici neboli riadky, ktoré by viedli k tomu, že by Algoritmus X našiel aj riešenia, ktoré môžu vzniknúť rotáciou iného riešenia.



Obr. 11: Symetrické riešenia hlavolamov.

Obrázok 11 ukazuje príklad dvoch riešení jednoduchého hlavolamu, ktoré sú symetrické - druhé sa dá vytvoriť z prvého tak, že sa otočí o 180 stupňov. Metóda *addSymmetries* má takúto štruktúru:

```
private fun addSymmetries(shape: Shape) {
    if (shape == shapeWithMostUniqueRotations) {
        val puzzleCopy = task.puzzle.copy()
        for (rotationMatrix in rotationMatrices) {
            puzzleCopy.rotate(rotationMatrix)
            shape.rotate(rotationMatrix)
            val offset = getMinCoord(puzzleCopy.points).invert()
            puzzleCopy.translate(offset)
            shape.translate(offset)
            if (task.puzzle == puzzleCopy) {
                symmetricShapes.add(shape.copy())
            }
        }
    }
}
```

V atribúte *shapeWithMostUniqueRotations* sa nachádza útvar, zo zadaných útvarov, ktorý má najviac unikátnych rotácií. Tento atribút sa nastaví ešte než sa začne vytvárať binárna matica problému. Keď metóda dostane tento tvar, zostaví všetky rotované uloženia tohto tvaru a pridá ich to množiny *symmetricShapes*. Metóda *isShapeGoodToBeAdded*, ktorá kontroluje či sa útvar nachádza vnútri priestoru hlavolamu aj kontroluje či sa dané posunutie

rotácie útvaru už nenachádza v množine *symmetricShapes*. Ak áno, nepridá sa do matice. Tento algoritmus funguje iba v prípade útvarov, ktoré v hlavolame musia byť umiestnené presne raz. Pre tie, ktoré môžu byť umiestnené ľubovoľne veľakrát sa toto použiť nedalo.

3.4. Určovanie náročnosti hlavolamov

V kapitole 2.5 som napísal, že môj program určuje náročnosť hlavolamov. Toto som implementoval pomocou triedy **Difficulty**, ktorá má metódu *calculateDifficulty*, ktorej návratová hodnota je náročnosť hlavolamu: číslo od 0 do 1. Algoritmus, ktorý náročnosť ráta je implementovaný takto:

```
fun calculateDifficulty(task: Task) : Double {
    val numberOfSolutions = solve(task)
    if (numberOfSolutions == 0) {
        return 0
    }
    searchSolutionForShapeFrequency()
    return calculateBaseDifficulty(task)  /
            numberOfSolutions()
}
```

Najprv sa hlavolam zadaný objektom **Task** vyrieši. Ak riešenie nemá, výsledok je, že hlavolam má náročnosť 0. Ak riešenie má, metóda *searchSolutionForShapeFrequency* prejde všetky riešenia hlavolamov a pre každý tvar, ktorý v hlavolame je, zistí, koľkými rôznymi spôsobmi je v riešeniach uložený.

Následne metóda *calculateBaseDifficulty* pre každý útvar vyráta podiel počtu rôznych uložení útvaru v riešeniach vyrátaných metódou a počtu všetkých možných uložení útvaru v priestore hlavolamu. Podiely prislúchajúce ku všetkým útvarom sa navzájom vynásobia a výsledné číslo vyjadrujúce celkovú šancu vyriešenia hlavolamu je výsledok metódy.

Nakoniec sa výsledok *calculateBaseDifficulty* vydelí počtom riešení hlavolamu. Takto vznikne finálne číslo vyjadrujúce pravdepodobnosť toho, že človek náhodne nájde riešenie hlavolamu.

3.5. Implementácia hry Genius Square

Komponent **geniusSquare** predstavuje implementáciu hry Genius Square. V jeho vnútri sú reprezentácie hracích kociek, ktorými sa v tejto hre hádže, útvary, ktorými sa v tejto hre vypĺňa hracia plocha a metódy na riešenie zadaní hry a taktiež na tvorbu nových zadaní.

Riešenie náhodného zadania

Trieda **GeniusSquare** poskytuje metódu *solveRandomGame*. Metóda z možných zadaní hry vyberie náhodne jedno a vyrieši ho.

Riešenie užívateľom vybraného zadania

Metóda *solveSpecifiedGame* má parameter *indices* slúžiaci na zadanie hodu kociek určujúcich polohu fixných valčekov. Metóda zistí, či taký hod mohol v tejto hre nastať, a ak áno, tak zadanie špecifikované týmto hodom vyrieši.

Riešenie všetkých zadaní

Niektoré zadania majú viac riešení, iné menej. Jednou z úloh, ktoré som riešil v rámci hry Genius Square bolo vyriešiť všetkých 62208 možných zadaní a spraviť nejakú štatistiku - minimálny počet riešení zadania, maximálny počet, priemerný počet, ale aj rozloženie počtu riešení, teda zistenie toho, ako často sa vyskytujú zadania s počtom riešení v určitých rozmedziach. Metóda *solveAllGames* vykoná presne túto prácu - vyrieši všetky možné zadania a vypíše štatistiky z tohto výpočtu.

Generovanie novej série zadaní

Posledná funkcionalita modulu je generovanie nových hracích kociek. Metóda *createNewGames* vygeneruje nové kocky s iným rozložením fixných bodov. Pôvodné hracie kocky hry Genius Square mali tú vlastnosť, že každý bod hracej plochy bol na nejakej kocke presne raz. Keďže sedem kociek má dokopy 42 strán a hracia plocha má len 36 dielikov, niektoré pozície sa v rámci kocky opakujú viackrát. Tieto opakovania som sa rozhodol ignorovať. Platí teda, že každá kocka má na svojich stenách 1 až 6 rôznych polôh z hracej plochy.

Pri generovaní nových hier, teda generovaní iného rozmiestnenia bodov z hracej plochy, ktoré umožní aj iné zadania tejto hry, som sa rozhodol tieto pravidlá tiež dodržovať. Použil som stratégiu, pri ktorej sa nové kocky vytvárajú z tých pôvodných. Generovanie prebieha takto:

```
private fun createNewGamesFromExistingGamesTactic() {
    if (depth == DEPTH_OF_CHANGE) {
        return
    }
    while (true) {
        val dicesCopy = copyDices(dices)
        movePosition(dicesCopy)
        if (doSolveAllGames(dicesCopy)) {
            depth++
            dices = dicesCopy
            return createNewGamesFromExistingGamesTactic()
        }
    }
}
```

Rekurzívna metóda *createNewGamesFromExistingGamesTactic* sa volá, kým nedosiahne hĺbku určenú konštantou *DEPTH_OF_CHANGE*. Pri každom volaní metóda v nekonečnom cykle vždy pomocou *movePosition* presunie jednu pozíciu hracej plochy z náhodne vybranej kocky na inú náhodne vybranú kocku, kde je tento presun možný. Potom otestuje, či pri tomto upravenom rozložení kociek platí, že všetky hry majú riešenie, a ak áno, rekurzívne sa vnorí s upravenými kockami. Metóda vlastne pomocou techniky pokus-omyl presúva pozície na kockách toľkokrát, koľko je hodnota *DEPTH_OF_CHANGE*. Čím väčšie toto číslo je, tým bude teoreticky výsledne rozloženie kociek rozdielnejšie od toho pôvodného.

3.6. Implementácia hry Ubongo 3D

Hru Ubongo 3D implementuje modul **ubongo**. Ten obsahuje konštanty veľkého množstva útvarov, ktoré sa v hre používajú a metódy na riešenie zadaní Ubongo 3D.

Riešenie užívateľom vybraného zadania

Pomocou metódy *solveSpecifiedGame* sa dá vyriešiť užívateľom zadané hry. Metóda má parametre pre popis miestnosti hlavolamu (objekt **Puzzle**) a pre množinu tvarov, ktorými sa bude pokrývať (objekt **Shape**). *SolveSpecifiedGame* hlavolam vyrieši a vypíše riešenia.

Generovanie nových zadaní

Generovanie zadaní Ubonga rieši metóda *solveGeneratedGame*. Táto dostane na vstupe celé číslo určujúce úroveň náročnosti hlavolamu a následne vygeneruje hlavolam, ktorý má danú náročnosť, vyrieši ho a vypíše jeho riešenia. Úrovne obťažnosti vyzerajú takto:

- 1 - ľahký hlavolam.
- 2 - stredne ťažký hlavolam.
- 3 - ťažký hlavolam.
- 4 - veľmi náročný hlavolam.

Generovanie zadaní Ubongo 3D určitej náročnosti, je naprogramované takto:

```
fun generateAndSolveGame(difficultyLevel: Int) {
    while (true) {
        checkConditions()
        generatePoint()
        solveGameRepeatable()
        findNonRepeatingSolution()
        val difficulty = difficultyOfSolution()
        if (difficulty == difficultyLevel) {
            break
        }
    }
    solveGame()
}
```

Metóda, ako bolo navrhnuté v kapitole 2.7, dookola skúša generovať priestory hlavolamov, následne pre tieto priestory zistiť, či existuje množina útvarov, ktoré ju zaplnia, a ak áno, overiť, či majú požadovanú náročnosť. Keď sa takéto zadanie hlavolamu nájde, generátor úspešne skonči svoj beh.

3.7. Vykresľovanie pomocou knižnice OpenGl

Na 3D renderovanie som použil triedy implementujúce interface **Printer**: **GLUbongoPrinter** a **GLGeniusSquarePrinter**. Prvá slúži na renderovanie riešení hier Ubongo 3D a druhá na hry Genius Square. Riešenia vykresľujú tak, ako bolo opísané v kapitole 2.8. Metóda *print* oboch tried vyzerá takto:

```
override fun print() {
    if (solutions.isNotEmpty()) {
        init()
        render()
        while (!glfwWindowShouldClose(window)) {
            glfwWaitEvents()
        }
    }
}
```

Ak hlavolam má nejaké riešenia, spustí sa sekvencia, ktorá najprv inicializuje okno, do ktorého sa bude kresliť. Následne sa vykreslí prvé riešenie hlavolamu a nakoniec sa v nekonečnom cykle čaká na vstupy od používateľa. S riešeniami hlavolamu sa dá manipulovať pomocou klávesnice a myšky:

- Tlačidlo “+“: priblíženie riešenia.
- Tlačidlo “-“: oddialenie riešenia.
- Pravá šípka: ďalšie riešenie hlavolamu.
- Ľavá šípka: predchádzajúce riešenie hlavolamu.
- Tlačidlo “ENTER“: animácia zloženia hlavolamu.
- Tlačidlo medzerník: prefarbenie riešenia náhodnými farbami.
- Ľavé tlačidlo na myške: odstránenie tvaru, na ktorý sa klikne.
- Pravé tlačidlo na myške: vrátenie posledného odstráneného tvaru.
- Tlačidlo “A“: otočenie okolo osi X.
- Tlačidlo “S“: otočenie okolo osi Y.
- Tlačidlo “D“: otočenie okolo osi Z.

Samotné vykresľovanie používa knižnicu OpenGL. Základná renderovaná jednotka je kocka. Algoritmus vykresľovania teda iba dookola kreslí kocky rôznych farieb na rôzne polohy tak, aby vytvoril 3D modely riešení hlavolamov.

4. Testovanie

Parametre zariadenia na ktorom som vykonával testy sú:

- Procesor AMD Ryzen 5 4500U.
- Operačná pamäť RAM DDR4 8 GB.
- Operačný systém Windows 10 Home.
- Úložisko typu SSD - 512GB.

4.1. Štatistika všetkých riešení hry Genius Square

Štatistiky riešení všetkých 62208 zadaní hry Genius Square ukázali, že minimálny počet riešení hry je 11, maximálny je 22317. Priemerný počet je zase približne 1936. Frekvencie počtov riešení sú takéto:

23180 hier má od 0 do 999 riešení.

18848 hier má od 1000 do 1999 riešení.

9018 hier má od 2000 do 2999 riešení.

4390 hier má od 3000 do 3999 riešení.

2576 hier má od 4000 do 4999 riešení.

1566 hier má od 5000 do 5999 riešení.

890 hier má od 6000 do 6999 riešení.

564 hier má od 7000 do 7999 riešení.

330 hier má od 8000 do 8999 riešení.

248 hier má od 9000 do 9999 riešení.

174 hier má od 10000 do 10999 riešení.

116 hier má od 11000 do 11999 riešení.

74 hier má od 12000 do 12999 riešení.

70 hier má od 13000 do 13999 riešení.

38 hier má od 14000 do 14999 riešení.

22 hier má od 15000 do 15999 riešení.

24 hier má od 16000 do 16999 riešení.

24 hier má od 17000 do 17999 riešení.

12 hier má od 18000 do 18999 riešení.

18 hier má od 19000 do 19999 riešení.

12 hier má od 20000 do 20999 riešení.

12 hier má od 21000 do 21999 riešení.

2 hry majú od 22000 do 22999 riešení.

Z toho je vidno, že viac ako polovica zadaní má menej ako 2000 riešení, pričom nad hranicou 10000 riešení je už len malé množstvo zadaní. Tento test zbehol za 2 hodiny a 18 minút.

4.2. Štatistiky generovania nových kociek hry Genius Square

Vygenerovanie iného rozloženia kociek z tých kociek, ktoré sú v hre, tak aby platilo, že každý hod vytvorí zadanie s riešením trvá:

DEPTH_OF_CHANGE * T

Číslo DEPTH_OF_CHANGE vyjadruje koľko presunov hodnôt z nejakej kocky na nejakú inú kocku sa má uskutočniť. Toto číslo je konštanta v kóde, ktorá má hodnotu 3, aby sa dosiahla väčšia odlišnosť od originálnych kociek, ktoré sú v tejto hre. Samozrejme hodnota sa dá v kóde zmeniť.

T vyjadruje priemerný čas v sekundách, za aký sa podarí uskutočniť jeden náhodný úspešný presun hodnôt medzi kockami. Tento čas sa pohybuje od 138 sekúnd do 706 sekúnd, pričom priemerná hodnota sa hýbe okolo 334 sekúnd (5 minút).

4.3. Štatistiky generovania nových zadaní hry Ubongo 3D

V Závislosti od úrovne náročnosti náhodne vygenerovaného hlavolamu, trvá generovanie hlavolamu takéto časy:

- Úroveň 1: od 0.41 sekundy do 1.48 sekundy. Priemerný čas je 0.54 sekundy.
- Úroveň 2: od 0.52 sekundy do 5.12 sekundy. Priemerný čas je 0.91 sekundy.
- Úroveň 3: od 1.85 sekundy do 27.81 sekundy. Priemerný čas je 8.17 sekundy.
- Úroveň 4: od 62.27 sekundy do 640.54 sekundy. Priemerný čas je 154.72 sekundy.

4.4. Benchmark pre Algoritmus X

Aby som zistil ako moja implementácia obstojí v porovnaní s implementáciami v iných programovacích jazykoch, s použitím iných dátových štruktúr, vyhotovil som si benchmark. Porovnávať budem moju implementáciu spravenú v jazyku Kotlin s implementáciou v jazyku Java [11] a implementáciou v jazyku C++ [12]. Testy boli vykonané na viacerých maticiach, ktoré vznikli formalizáciou rôznych zadaní hier Genius Square. Priemerné výsledky testovania vyzerajú takto:

Kotlin	Java [11]	C++ [12]
94 ms	29 ms	42 ms
1218 ms	190 ms	619 ms
66 ms	3 ms	7 ms
67 ms	16 ms	116 ms
87 ms	10 ms	37 ms
204 ms	14 ms	72 ms
277 ms	38 ms	128 ms

Na tabuľke je vidno, ako moja implementácia obstála v porovnaní s jazykmi, ktoré sú poznateľne rýchlejšie a výkonnejšie. C++ implementácia je v niektorých prípadoch

značne efektívnejšia, v niektorých dokonca pomalšia. Toto je z dôvodu, že používa podobné dátové štruktúry na ukladanie matíc ako ja. Java implementácia porazila moju pri každom teste, dosahuje približne 8 až 10 násobnú rýchlosť tej mojej. Za dôvod tohto výsledku považujem fakt, že implementácia na reprezentáciu matíc používa rýchlejšie dátové štruktúry, ktorými sú polia.

Z benchmarku sa dá posúdiť, že Kotlin implementácia relatívne obstála v porovnaní s inými jazykmi, a že zmenou dátových štruktúr používaných na najnižších úrovniach by bolo možné dosiahnuť ešte väčšie priblíženie ku ich výkonom.

Záver

Moja implementácia Algoritmu X je použiteľná ako samostatná knižnica. Taktiež bola otestovaná na veľkom množstve vstupov a aj úspešne porovnaná s inými implementáciami. V tomto porovnaní dosahovala dobré výsledky.

Taktiež viem riešiť ľubovoľné zadania hier Ubongo 3D a Genius Square, pričom program nájde len jedinečné riešenia, teda žiadne také, ktoré by sa dali vytvoriť symetrickým zobrazením z iného riešenia. Okrem tohto je môj program natoľko všeobecne postavený, aby bol jednoducho rozšíriteľný o iné typy hlavolamov, napríklad aj o hypotetické N-rozmerné hlavolamy.

V mojej práci som sa nielen zameriaval na riešenie takýchto hlavolamov, ale čo je viac podstatné, snažil sa čo najpresnejšie porovnať ich náročnosť a určiť, čo presne robí hlavolam náročným. Na základe tohto porovnania som vytvoril aj generátory, ktoré vedia vytvoriť hlavolamy rôznych náročnosti, vrátane veľmi náročných hlavolamov.

Jednotlivé riešenia hlavolamov umožňujem zobraziť jednoducho v konzole, ale aj pomocou 3D renderovania s použitím knižnice OpenGL. S riešeniami sa dá manipulovať pomocou klávesnice na počítači.

V neposlednom rade som sa kód snažil pomocou neustálej refaktorizácie udržovať v čistom stave. Takto som chcel dosiahnuť to, aby kód obsahoval čo najmenej neočakávaných chýb a nedostatkov. Taktiež to umožňuje jednoducho pochopiť jeho funkcionalitu a aj rozšírenie o novú funkcionalitu.

Zadanie mojej práce som verím úspešne splnil a popasoval som sa so všetkými nástrahami, ktoré sa popritom objavili. Táto práca ma naučila základy 3D renderovania a prehĺbila zásady čistého kódu, ktoré ovládam.

Práca má podľa môjho názoru ešte potenciál na viaceré zlepšenia. Optimalizovať by sa dali viaceré procesy vrátane generovania hlavolamov a samotného Algoritmu X, kde by sa dalo dosiahnuť veľké zrýchlenie použitím iných dátových štruktúr. Odstraňovanie symetrických riešení hlavolamov by sa tiež dalo zefektívniť. Moja práca by sa tiež dala použiť ako základ pre interaktívne hry, ktoré by umožnili používateľovi, aby sám riešil hlavolamy, ktoré by môj program vygeneroval. Riešenia používateľa by môj program následne vedel aj skontrolovať.

Použitá literatúra

[1] Donald E. Knuth. *The Art of Computer Programming, Volume 4*. Addison-Wesley, 1968.

[2] Donald E. Knuth. *Dancing Links*. In Millenian Perspectives in Computer Science, strany 187-214, 2000.

[3] Mattias Harrysson, Hjalmar Laestander. *Solving Sudoku efficiently with Dancing Links.* Stockholm, 2014.

[4] Éva Tardos, Jon Kleinberg. *Algorithm Design*. Addison-Wesley, 2005.

[5] Andrej Blaho. *Spájané štruktúry:* http://input.sk/python2016/26.html [Online, 25.5.2021].

[6] Andrej Blaho. *Spájané zoznamy:* http://input.sk/python2016/27.html [Online, 25.5.2021].

[7] JetBrains s.r.o. *Kotlin Documentation*: https://kotlinlang.org/docs/home.html [Online, 25.5.2021].

[8] JetBrains s.r.o. *IntelliJ Documentation*: https://www.jetbrains.com/help/idea/discover-intellij-idea.html [Online, 25.5.2021].

[9] Khronos Group. *OpenGL Documentation*: https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/ [Online, 25.5.2021].

[10] LWJGL. *LWJGL Documentation*: https://javadoc.lwjgl.org/overview-summary.html [Online, 25.5.2021].

[11] Benfowler. *Implementácia Algoritmu X v jazyku Java*: https://github.com/benfowler/dancing-links [Online, 25.5.2021].

[12] Jiaheng. *Implementácia Algoritmu X v jazyku C++*: https://github.com/jiaheng/sudoku-solver-cpp [Online, 25.5.2021].