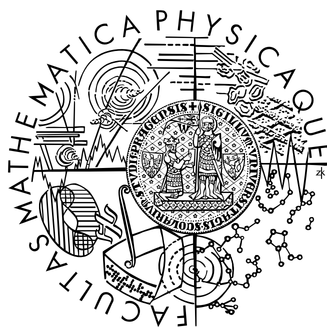
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Václav Dřízhal

SlitherLink

Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Jiří Dokulil

Studijní program: Informatika, Obecná informatika

2009

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

Václav Dřízhal

V Praze dne 3. 12. 2009

Obsah

1 Úvod 6

2 Metody pro řešení puzzle 7

2.1 Strategie řešení 7

2.2 Strategie založená na pravidlech 7

2.3 Zámky a CAL zámky 9

2.4 Obecné schéma algoritmu pro řešení puzzle 10

2.5 Pravidla založená na CAL zámcích 12

2.6 Normální zámky v políčku 15

2.7 Jednoduchá kontrola krátkých cyklů 16

2.8 3-zámky 19

2.9 Diagonální zámky 20

2.10 Obarvování políček 21

2.11 Obarvování hran 27

2.12 Pokročilé dedukce krátkých cyklů nad obarvením hran 30

2.13 Pokročilé barevné dedukce v políčku 32

2.14 Pokročilé barevné dedukce nad hranami v políčku 34

2.15 Pokročilé dedukce cest 34

2.16 Vylepšená propagace pravidel v políčku 41

3 Úrovně obtížnosti 43

3.1 Jednotlivé úrovně 43

3.2 Obecně k obtížnosti puzzle 45

3.3 Highlander dedukce 45

4 Generátor puzzle 47

4.1 Generování smyčky 47

4.2 Generování puzzle 51

5 Implementace 53

5.1 Použitá technologie 53

5.2 Reprezentace hrací plochy 53

5.3 Prioritní fronta 54

5.4 Popis puzzle pomocí řetězce Loopy 54

5.5 Schéma algoritmu pro řešení puzzle 55

6 Grafické uživatelské rozhraní 57

7 Možná rozšíření 59

7.1 Dedukce na základě cyklů obarvení políček 59

7.2 Obecný tvar políček 59

8 Závěr 60

9 Literatura a zdroje 63

10 Dodatky 64

10.1 SL Pro puzzle 64

10.2 Demonstrace dedukcí CAL zámků 64

Nedílnou součástí práce je také přiložené CD s programem.

Název práce: SlitherLink
Autor: Václav Dřízhal
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí bakalářské práce: RNDr. Jiří Dokulil
e-mail vedoucího: Jiri.Dokulil@mff.cuni.cz

Abstrakt: Cílem práce je vytvořit program, který bude sloužit jako pomůcka pro řešení hry SlitherLink na počítači. Program bude podporovat generování a řešení jednotlivých map hry a bude poskytovat komfortní grafické rozhraní usnadňující řešení: ovládání myší i klávesnicí, barvení, hledání oblastí, inteligentní víceúrovňové Fix/Undo, ukládání hry a export/import do Loopy formátu. Důraz bude kladen i na rychlost algoritmů řešení. Řešení bude probíhat pomocí předem připravené množiny pravidel a heuristik (barvení, zámky, apod.).

Klíčová slova: Slitherlink, logická hra, počítačová hra

Title: SlitherLink
Author: Václav Dřízhal
Department: Department of Software Engineering
Supervisor: RNDr. Jiří Dokulil
Supervisor's e-mail address: Jiri.Dokulil@mff.cuni.cz

Abstract: The aim of thesis is to create a program that would allow the user to solve SlitherLink puzzles. The program should support automatic generation an solving of the levels and it should provide user friendly graphical user interface: keyboard and mouse controls, coloring, finding of areas, intelligent multilevel Fix/Undo, load/save and import/export to Loopy format. The puzzle solving implementation should be efficient and it should utilize predefined set of rules and heuristics (coloring, locks etc.)

Keywords: Slitherlink, logic game, computer game

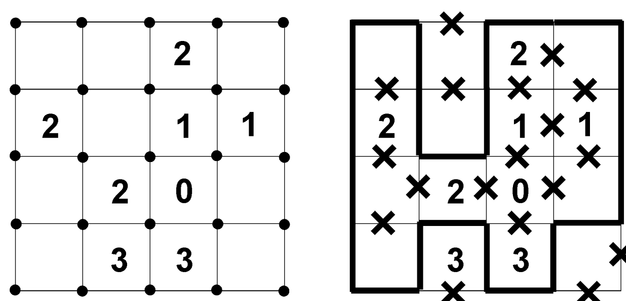
1 ÚVOD

Jiné názvy pro tuto hru jsou také Fences, Takegaki, Loop the Loop, Loopy, Ouroboros, Suriza nebo Dotty Dilemma.

Pravidla

Slitherlink je logická hra. Hraje se na obdélníkové hrací ploše obsahující tečky, které budeme nazývat mřížové body. Ty nám vymezují políčka, která mohou, ale nemusí, obsahovat číselný údaj. Pokud políčko obsahuje nějaké číslo, z rozsahu 0-3, potom ho nazveme nápovědou. Cílem hry je vytvořit na hrací ploše jedinou smyčku propojující sousední mřížové body bez toho, aby se tato smyčka někde křížila či větvila. Nápověda nám určuje, kolika hranami smyčka dané políčko obklopuje. Všechny nápovědy musejí být splněny. Navíc řešení by mělo být jednoznačné.

Následující obrázek (1.1) nám znázorňuje instanci Slitherlinku (zkráceně budeme nazývat puzzle) a jeho řešení



Obrázek 1.1 – příklad puzzle a jeho řešení

Historie

Dle wikipedie [1] toto puzzle pochází od Nikoli, velmi známé japonské firmy s logickými puzzle. Puzzle se poprvé objevilo v červnu roku 1989 ve vydání Puzzle Communication Nikoli #26. Vzniklo tak, že editor zkombinoval dva různé druhy puzzle dohromady. Původně obsahovalo každé políčko nápovědu.

NP-úplnost

Je dokázáno, že tato hra je ve třídě NP-úplných úloh (viz článek v japonštině od T. Yato [2])

2 METODY PRO ŘEŠENÍ PUZZLE

2.1 STRATEGIE ŘEŠENÍ

Strategie řešení Slitherlinku není nikde příliš popsána a neexistují obecné principy pro řešení složitějších instalací jako je tomu například u Sudoku (např. Hidden Single, Swordfish pattern, X-Wing pattern). Tento nedostatek mne vedl později k vymyšlení vlastních postupů.

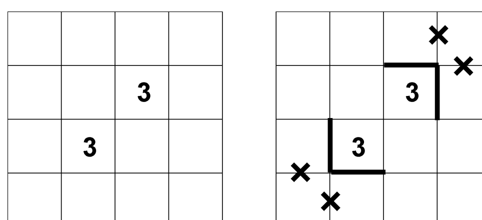
2.2 STRATEGIE ZALOŽENÁ NA PRAVIDLECH

Řešit instalaci Slitherlinku čistě backtrackingem je z hlediska časové efektivity algoritmu nemyslitelné (zejména pro větší instance). Proto přicházejí na řadu pravidla. Tato metoda je zejména oblíbená u začátečníků, kteří hrají tuto hru. Je to také téměř jediná metoda, která je dobře popsána. V této kapitole vycházím z textu článku pocházející od Hertinga [3].

Pravidla se skládají ze dvou částí:

1. předpoklad – množina hran a nápověd políček v určitém seskupení
2. důsledek – množina hran plynoucí k předpokladu (je větší ve smyslu inkluze než množina hran předpokladu)

Princip pravidel spočívá v tom, že pokud na nějakém místě hrací plochy nalezneme předpoklad, tak ho rozšíříme na důsledek uplatněním pravidla. Tímto způsobem je možné, že nám vznikne předpoklad na novém místě a řetězově postupujeme do té doby, dokud lze uplatňovat pravidla.



Obrázek 2.2 – příklad pravidla

Výhody a nevýhody této metody

Zřejmě platí, že čím více pravidel řešitel (počítač) zná, tím více je toho schopen vyřešit. Existují dvě metody, jak vygenerovat množinu pravidel:

a) První metoda je napsat tuto množinu ručně. Tento způsob je však příliš pracný, náchylný na chyby (a jejich hledání) a tato množina je nekompaktní a závislá na autorovi.
b) Druhá metoda spočívá v tom, že na počítači vygenerujeme všechny možné předpoklady (omezené velikosti) tj. nápovědy a prázdná políčka a množinu učených hran. Ke každému takovému předpokladu *P* určíme dále všechna možná ohodnocení neurčených hran, označme tuto množinu jako *O*. Z množiny *O* vybereme pouze konzistentní prvky (tj. neporušující žádné ze základních pravidel Slitherlinku a označme tuto množinu jako *K*. Dále definujeme množinu hran *D* předpisem $D = \cap K$, neboli množina *D* obsahuje pouze hrany společné pro všechna *K*, tj. pro všechna konzistentní ohodnocení k předpokladu *P*. Nyní rozlišíme tři případy:

1) $D = \emptyset$ nebo $D \subseteq P \Rightarrow$ chybný předpoklad (nekonzistentní sám o sobě)
2) $D = P \Rightarrow$ nejde o pravidlo (nezjistili jsme nic nového)
3) $P \subseteq D \Rightarrow$ jde o pravidlo (vydedukované hrany jsou D \ P)

Zbývá však otázka, jakou velikost mají mít předpoklady. Pokud by měly velikost 1 políčko, tak nám předpoklady příliš nepomohou. Proto se sama nabízí možnost 2 x 2 políčka. Uvažme nejprve kolik je možnosti pro tuto množinu pravidel. Políčka mohou nabývat 5 možností (0, 1, 2, 3 a bez nápovědy), hrany 3 možností (čára, křížek a neurčená hrana). Jednoduchým výpočtem zjistíme, že počet všech předpokladů je $5^4 \,.\, 3^{12} \approx 332.10^6$. Značná část předpokladů z tohoto počtu však není konzistentní a další značná část netvoří pravidla. I přes tento fakt je množina pravidel stále příliš vysoká. Snížit lze třemi způsoby – odstraněním symetrií, dominovaných pravidel a složených pravidel.

I přes všechna tato vylepšení je stále množina pravidel dosti velká. A z tohoto důvodu je aplikace pravidel pomalá – na každé pozici hrací plochy musíme projít celou množinu pravidel, abychom zjistili, zda nelze nějaké uplatnit. Navíc pokud bychom odstranili složená pravidla, je třeba v případě nalezení (aplikace) pravidla, zkontrolovat tuto pozici znovu – je zde možnost uplatnit další pravidlo (nyní máme silnější předpoklad).

Dále při odstranění symetrií se změní pouze velikost paměti potřebné pro uložení pravidel. Vliv na efektivitu to nemá – v tomto případě musíme každé pravidlo všemi možnostmi pootočit a překlopit, což naopak vede ke snížení efektivity. Spousta pravidel je sama o sobě symetrických.

Můj první záměr tedy spočíval ve využití pravidel s tím, že dám přednost časové efektivitě před paměťovou. Rozhodl jsem se neodstraňovat redundantní pravidla a efektivitu naopak zvýšit tím, že jsem nad pravidly vybudoval vyhledávací strukturu tvaru stromu. Strom obsahoval kořen a pět synů (nápovědy u prvního políčka), každý z těchto synů měl dalších 5 synů (nápovědy u druhého políčka) a takto to pokračovalo až ke čtvrtému políčku, kde byli už jen 3 synové (stav u první hrany) a dále následovaly i ostatní hrany.

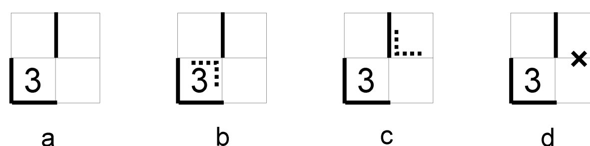
Hlavní výhoda tohoto principu spočívala v tom, že i přestože počet pravidel byl větší než 3 miliony, tak stačilo maximálně 16 porovnání (4 políčka a 12 hran) na jednu pozici. V případě, že byla možnost uplatnění pravidla, tak v koncovém uzlu vyhledávacího stromu byl nalezen důsledek (nejsilnější možný). V opačném případě nešlo ve vyhledávacím stromě postoupit dále a hledání bylo ukončeno předčasně. Podle provedených testů stačilo v průměru okolo 6 porovnání na jednu pozici. Dále jsem vymyslel kódování souboru s pravidly na bitové úrovni, které mi zajistilo kompresní poměr větší než 3 (pro uložení na disku).

Avšak i přes tato vylepšení byl tento princip „kostrbatý" a nedokázal postihnout globální aspekty hry (pracoval pouze na lokální úrovni pomocí těchto jednoduchých pravidel) a byl paměťově velmi náročný. Lepší propagaci do okolí umožňují například zámky

2.3 ZÁMKY A CAL ZÁMKY

Motivace

Zámkem se obecně nazývá vztah mezi dvojicí hran. Taktéž obecně se tento vztah definuje jako binární. Buď dvě hrany musejí mít stejnou hodnotu (obě křížky či obě čáry) nebo opačnou (právě jedna hrana musí být křížek a druhá čára nebo naopak). Zámky potom umožňují více dedukcí než pouze samotné hrany. Tuto techniku si předvedeme na obrázku 2.3.1.



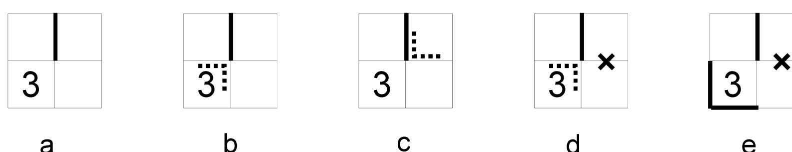
Obrázek 2.3.1 – demonstrace zámků

Předpokládejme situaci zachycenou na obrázku 2.3.1 v části a (jedná se pouze o výřez puzzle). Pokud se zaměříme na nápovědu s číslem 3, zjistíme, že dvě ze čtyř hran jsou neurčeny (znázorněny jsou čárkovaně v části b na obrázku 2.3.1). Dvě určené hrany jsou čarami, tudíž ke splnění nápovědy nám chybí již jediná čára. Z toho plyne, že v dvojici neurčených hran musí být právě jedna čarou a jedna křížkem. Jinými slovy musejí být opačné. Nyní se zaměřme na mřížový bod ve středu výřezu puzzle. Již víme, že dvojice hran, která se dotýká nápovědy 3, musí být opačná. Zaměříme se nyní na zbylé dvě hrany v tomto mřížovém bodě (znázorněny jsou opět čárkovaně na obrázku 2.3.1 v části c). Snadno dospějeme k tomu, že musejí být taktéž opačné. Kdyby tomu tak nebylo, mohou nastat dvě možnosti. První možností je, že obě budou křížky (v situaci znázorněné na obrázku 2.3.1 je však tato možnost vyloučena přítomností čáry). Potom bychom dospěli k uvěznění čáry a nikdy bychom smyčku už neuzavřeli. Druhou možností je, že jsou obě čáry. V této situaci bychom dostali větvení v mřížovém bodě, což je pravidly zakázáno. Jelikož víme, že tedy tato dvojice hran musí být opačná a zároveň víme,

že jedna z hran je čára, pak z toho ihned plyne, že druhá hrana musí být křížek (část d na obrázku 2.3.1).

Zavedení CAL zámků

Pro více dedukcí a lepší propagaci jsem zavedl *CAL zámky*. Rozdíl mezi klasickými zámky je v tom, že nenabývají dvou stavů, nýbrž tří. Jedná se v podstatě o jemnější klasifikaci dvojice hran – zda mohou být obě hrany křížky (Crosses), nebo obě mohou být opačné (Antilocked), či obě mohou být čarami (Lines). Podívejme se na jejich výhodu oproti klasickým zámkům.



Obrázek 2.3.2 – demonstrace CAL zámků

Předpokládejme situaci naznačenou na obrázku 2.3.2 v části a. Pomocí klasických zámků nelze vydedukovat nic. Zaměřme se na dvojici hran zobrazenou čárkovaně v části b. Jelikož obě tyto hranou sousedí s políčkem obsahujícím nápovědu 3, vyplývá z toho, že povolený zámek mezi nimi je AL. Hodnotu C z triviálních důvodů obsahovat nemůže. Zaměřme se nyní na druhou dvojici hran u mřížového bodu ve středu výřezu puzzle (zvýrazněny jsou na obrázku 2.3.2 v části c). Jelikož jedna hrana z nich je již určena (čára), pak zámek mezi nimi musí být AL. Při zamyšlení zjistíme, že hodnota L u tohoto zámku je nepřístupná. Pokud by byla L, pak by první dvojice hran musela být C (v mřížovém bodě je buď 0 čar, nebo právě 2 – jiné počty nejsou přípustné). Hodnotu C jsme ale u ní vyloučili. Tudíž jako jediný kandidát zbývá zámek A a ve spolupráci s tím, že jedna z hran je čára, musíme označit druhou křížkem (část d). Navíc nám to ovlivní první dvojici hran – přípustná bude hodnota A – a ta nám v políčku s nápovědou 3 implikuje dvě zbylé čáry (obrázek 2.3.2 část e). Další příklady dedukcí CAL zámků lze nalézt v Dodatku (10.2).

Později uvedu i další druhy zámků, které jsem zavedl. Liší se v tom, že nemusejí být definované nad přímo sousedícími hranami, nebo že jsou definovány nad jiným počtem hran (viz kapitola 3-zámky).

2.4 Obecné schéma algoritmu pro řešení puzzle

Motivace

Při řešení slitherlinku obvykle postupujeme tak, že vydedukujeme na nějakém místě hranu a z toho plynou další dedukce v jejím okolí. Nevhodný je

proto přístup založený na procházení celé hrací plochy v každém kroku algoritmu a hledání místa, kde lze něco vydedukovat. Představme si například situaci, kdy vydedukujeme hranu v posledním řádku hrací plochy, ze které vyplývá například dedukce v políčku v předposledním řádku (díky výše popsanému lokálnímu charakteru hry). Potom musíme zahájit další krok spočívající ve zbytečné kontrole spousty mřížových bodů a políček, přestože tam nelze nic vydedukovat. Takovýmto způsobem je například implementována aplikace Loopy.

Idea algoritmu

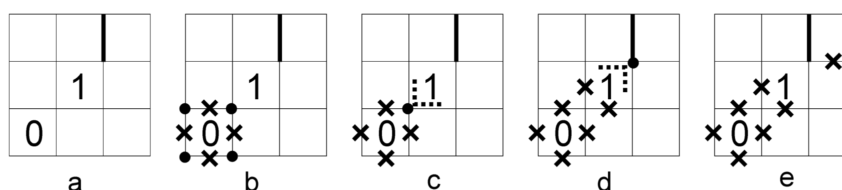
V algoritmu budeme používat prioritní frontu popsanou v kapitole 5.3. Přesněji řečeno budeme používat fronty dvě – jedna bude pro mřížové body a jedna pro políčka. Při inicializaci vložíme do fronty políček pouze políčka s nápovědou a do fronty mřížových bodů takové mřížové body, jenž leží na hranici hrací plochy (je to z toho důvodu, že v nich je určena alespoň jedna hrana – ta, co leží mimo hrací plochu).

Pro každý mřížový bod si uchováváme příznak, zda již není vyřešen (tj. určeny všechny 4 hrany). Je to mechanismus, který nám zajišťuje, abychom nededukovali zbytečně. Pokud totiž dostaneme mřížový bod k dedukcím, nejprve otestujeme, zda není již kompletní, a v kladném případě se jím nezabýváme. Umožňuje to jednak urychlení výpočtu a za druhé konečnost algoritmu (vyčerpání front). Analogicky si uchováváme příznaky i pro políčka.

Mřížové body mají menší počet možných stavů (7 stavů), kterých mohou nabývat. Proto je lepší mřížové body zpracovávat dříve než políčka. Políčka dělíme do několika kategorií (priorit) podle jejich časové složitosti na zpracování. Nejdříve zpracováváme políčka, která jsou mimo hrací plochu (od začátku mají příznak, že jsou kompletní). Dále zpracováváme nápovědu 0, počet možných stavů je 16, ale po jednom průchodu máme jistotu, že políčko již bude splněno (určíme všechny 4 hrany okolo). Poté zpracováváme po řadě nápovědu 3 (16 stavů), nápovědu 1 (64 stavů), nápovědu 2 (64 stavů) a nakonec až políčko bez nápovědy (160 stavů).

Vzájemné ovlivňování políček a mřížových bodů

Nejlépe se metoda ovlivňování předvede na obrázku (jedná se pouze o výřez z puzzle).



Obrázek 2.4 – ovlivňování políček a mřížových bodů

Předpokládejme situaci naznačenou na obrázku 2.4 v části a. Dále předpokládejme, že z fronty políček vyzvedneme políčko s nápovědou 0.

Provedeme nad ním dedukce (aplikaci pravidel), přičemž dostaneme 4 křížky. Zároveň se nám ale změnil stav všech 4 mřížových bodů (díky změněným hranám). Situaci zachycuje obrázek 2.4, část b. Mřížové body vložíme tedy do fronty mřížových bodů. Jak bylo řečeno dříve, nejprve pracujeme s mřížovými body kvůli efektivitě algoritmu. Nechť vyzvedneme z fronty mřížový bod označený na obrázku 2.4 v části c. Provedeme nad ním dedukce a zjistíme, že zbývající dvojice neurčených hran (naznačena tečkovaně) musí mít stejnou hodnotu (obě buď křížky, nebo čáry – odpovídá to zámku CL). Toto je nová skutečnost, ze které můžeme dále dedukovat, a to v políčku s nápovědou 1. Po vyčerpání všech mřížových bodů se podíváme, zda máme nějaký prvek ve frontě políček. Vyzvedneme si políčko s nápovědou 1 a provedeme nad ním dedukce. Zjistíme, že čárkovaně naznačené hrany musejí být křížky, jinak bychom porušili nápovědu. Situace je zachycena na obrázku 2.4 v části d. Zbylé dvě hrany v políčku s nápovědou musejí mít opačnou hodnotu (právě jedna musí být hrana a jedna křížek – odpovídá to zámku A). Opět je to pro nás nová skutečnost a ovlivní nám to mřížový bod zvýrazněný na obrázku. Po provedení dedukcí nad ním dospějeme k jednomu křížku (obrázek 2.4 část e).

Obecné schéma algoritmu

Schéma vypadá následovně.

dokud je alespoň jedna fronta neprázdná
{
    dokud je fronta mřížových bodů neprázdná
        vyzvedni mřížový bod a proveď dedukce
    jestliže je fronta políček neprázdná
        vyzvedni políčko a proveď dedukce
}

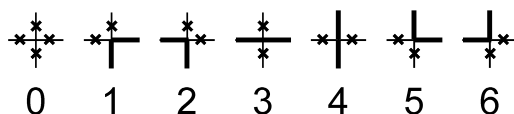
Povšimněme si toho, že je zde snaha upřednostňovat mřížové body. Dokud není fronta mřížových bodů prázdná, tak je zpracováváme. V případě, že se nám vyprázdní, tak se přesuneme ke frontě políček. Rozdíl spočívá v tom, že vyzvedneme pouze jediné políčko, nad kterým provedeme dedukce. Je zde totiž velká pravděpodobnost, že po provedení dedukcí máme opět frontu mřížových bodů neprázdnou. Výrazně tato implementace zvyšuje efektivitu algoritmu. Ještě pro doplnění políčka se vyzvedávají z fronty dle priorit (jak bylo nastíněno výše v textu). Nejprve tedy kontrolujeme ta políčka, kde předpokládáme nejméně práce.

2.5 Pravidla založená na CAL zámcích

Motivace

Pravidla zavádíme pro usnadnění spolupráce hran se zámky, zároveň představují určité zrychlení algoritmu. Situaci si předveďme na mřížových bodech. Pokud bychom pracovali bez pravidel, museli bychom při dedukci zkoumat kolik hran je určených, dále které to přesně jsou z hlediska umístění, dále jaké zámky jsou určeny, kde se nacházejí a jaká je jejich hodnota. Z hlediska implementace by to

představovalo několik vnořených rozhodovacích klauzulí (switch) a nízkou čitelnost kódu. Rovněž by bylo obtížné dělat případné modifikace a hledat chyby v této implementaci. Přitom u mřížových bodů existuje pouze 7 korektních stavů (viz obrázek 2.5.1).

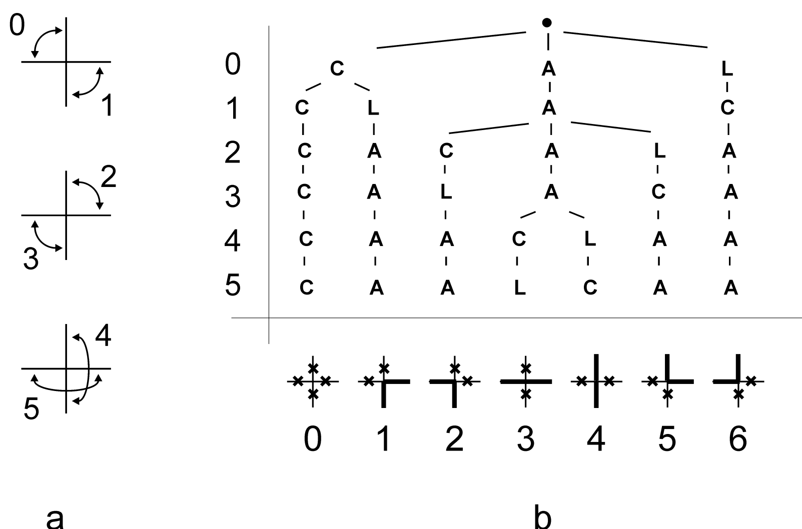


Obrázek 2.5.1 – přípustné možnosti mřížových bodů

Pravidla mřížových bodů a práce s nimi

Reprezentace

Pro každý mřížový bod si pamatujeme dva údaje. Jedním jsou zámky, druhým přípustné možnosti. Pokud jde o zámky, tak uvažujeme CAL zámek nad každou možnou dvojicí hran (viz obrázek 2.5.2 část a). V části b na témž obrázku pak lze vyčíst jednotlivé zámky k dané přípustné možnosti mřížového bodu.



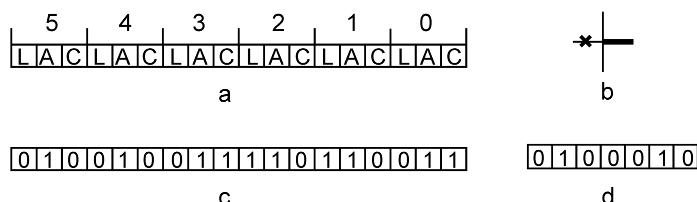
Obrázek 2.5.2 – popis mřížových bodů pomocí zámků

Jeden zámek reprezentujeme pomocí trojice bitů, symbolizujících po řadě C, A, L. Pokud je některý z bitů nastaven na hodnotu 1, pak to znamená, že zámek může nabývat danou hodnotu. Na počátku, před veškerými dedukcemi, je C = 1, A = 1, L = 1 pro každý zámek. Algoritmus pracuje tak, že vyřazuje nevhodné hodnoty zámků a možnosti, které by porušily konzistentnost puzzle.

Podívejme se nyní na obrázek 2.5.3. Nejprve v části a je vyobrazena struktura zámků u mřížového bodu, tak jak ji reprezentujeme. Jednotlivá políčka

představují bity určující, zda v daném zámku je daná hodnota přípustná. Tuto strukturu reprezentujeme pomocí jednoho 32bitového čísla.

Pro objasnění uvedeme příklad. Na obrázku 2.5.3 v části b je znázorněn mřížový bod, kde dvě hrany jsou již určeny. Odpovídající přípustné hodnoty zámků lze nalézt v části c téhož obrázku. Nakonec v části d nalezneme přípustné možnosti (stavy) mřížového bodu.



Obrázek 2.5.3 – ukázka zámků a stavů mřížového bodu

Práce s pravidly u mřížových bodů

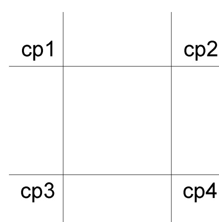
Při dedukcích postupujeme následovně. Nejprve načteme množinu přípustných stavů a aktualizujeme ji tím, že se podíváme na 4 okolní hrany. Vždy, pokud je nějaká hrana určena, provedeme restrikci přípustných stavů v závislosti na typu hrany a umístění. Dejme tomu, že máme mřížový bod z obrázku 2.5.3 v části b. Před spuštěním dedukcí byly přípustné všechny stavy (tj. 1111111). Po objevení křížku provedeme restrikci a dostaneme jako přípustné stavy 0, 1, 4, 5 (tj. 0110011). Poté objevíme čáru a provedeme opět restrikci a dostaneme již jen stavy 1, 5 (tj. 0100010).

Poté aplikujeme pravidla založená na zámcích. Snažíme se však aplikovat pouze taková pravidla, která připouštějí přípustné stavy mřížových bodů, ostatními se nezabýváme. Kontrola přípustnosti jednoho pravidla pak spočívá pouze v logickém součinu (AND) stavu zámků v mřížovém bodě a pravidla. Pokud logický součin je roven pravidlu, znamená to, že pravidlo lze aplikovat, neboť jedničkové bity pravidla jsou podmnožinou stavu zámků. Zároveň s tímto udržujeme největší možnou množinu společných hran všech přípustných pravidel. Pokud je po aplikaci pravidel tato množina větší než počet dosud známých hran, pak jsme vydedukovali stav alespoň jedné nové hrany. Jestliže jsou určeny všechny 4 hrany, pak prohlásíme mřížový bod za kompletní a již se jím nezabýváme. V opačném případě aktualizujeme hrany, pokud byly nějaké nové vydedukované (a s tím zároveň přidáme do fronty políček vždy ta dvě políčka, která s touto hranou sousedí – je zde vysoká pravděpodobnost nových dedukcí). Rovněž projdeme první 4 zámky. Pokud se některý z nich změnil, může to znamenat další dedukce v políčku, které sdílí dvě hrany tvořící tento zámek.

Díky tomu, že aplikace pravidel je založená na logickém součinu (pravidlo je zakódováno do jediného čísla), algoritmus pracuje velice rychle.

Pravidla políček

Pravidla políček popíšeme pouze obecně, neboť uvedeme více druhů pravidel v dalších kapitolách. Veškerá pravidla však mají společnou bázi. Tou je čtveřice mřížových bodů obklopujících políčko (přesněji řečeno množiny jejich přípustných stavů). Další informace záleží na tom, jakou skutečnost se snažíme zachytit pomocí pravidel. Tatáž skutečnost lze totiž popsat z různých úhlů pohledu a umožňuje tak rozdílné dedukce. Celkový počet pravidel je 160, což je relativně malá množina. Obsahují veškeré korektní situace 12 hran obklopujících políčko (viz obrázek 2.5.4). Pokud bychom uvažovali pouze 4 hrany okolo políčka, nedosáhli bychom téměř žádné propagace a komunikace s okolními políčky.



Obrázek 2.5.4 – obecné schéma pro pravidla v políčku

Podotkněme, že pravidla se aplikují opět pomocí logického součinu. Předtím ještě aktualizujeme stav ze stavů okolních mřížových bodů. Vydedukované změny nám ovlivňují okolní mřížové body a políčka.

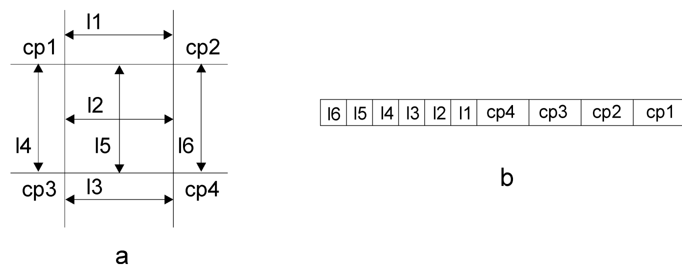
2.6 NORMÁLNÍ ZÁMKY V POLÍČKU

Motivace

Jedná se o první implementovaný set pravidel, který umožňuje práci zároveň s hranami i se zámky. Umožňuje však více než klasické dedukce založené na zámcích. Je to způsobeno tím pravidla nám omezují možné přípustné stavy jednotlivých mřížových bodů a naopak. Dále k tomu přispívají také zámky mezi příčnými hranami.

Princip

Obrázek 2.6 (část a) zachycuje strukturu normálních zámků. Pravidla jsou složena z přípustných možností mřížových a z CAL zámků mezi příčnými hranami. To umožňuje lepší interakci mezi políčky. Nechť například zjistíme, že po aplikaci pravidel se zámek l1 změnil a jeho hodnota je nyní CL. Potom z toho plyne, že v políčku bezprostředně umístěným nad tímto políčkem musí být zámek l2 také nastaven na hodnotu CL.



Obrázek 2.6 – struktura a reprezentace diagonálních zámků

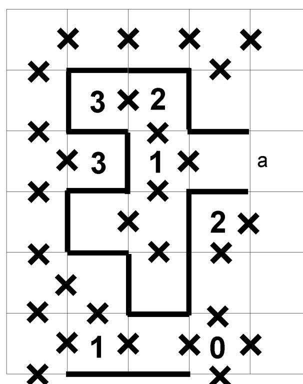
Velikost jednoho pravidla je 4 x 7 + 6 x 3 = 46 bitů. Pro reprezentaci jednoho pravidla nám tedy stačí využít 64bitové číslo. Způsob této reprezentace je zachycen na obrázku 2.6 v části b.

Aktuální stav políček je potřeba uchovávat v poli.

2.7 JEDNODUCHÁ KONTROLA KRÁTKÝCH CYKLŮ

Motivace

Častokrát se stává v průběhu řešení puzzle, že nás od krátkého cyklu dělí pouze jediná hrana. Pro uživatele je na první pohled zřejmé, že takováto hrana musí být křížkem a umožňuje to ve velkém množství případů další dedukce.



Obrázek 2.7.1 – krátké cykly

Na obrázku 2.7.1 lze snadno vidět, že hrana a, pokud by byla nastavena jako čára, by nám vytvořila krátký cyklus (obrázek je pouze výřez části puzzle). Otázka spočívá v tom, jak se proti takovým situacím bránit. Nasnadě jsou dvě triviální řešení. Jednak bychom mohli začít zkoumat (trasovat) všechny souvislé segmenty právě tehdy, když již nelze nic vydedukovat ostatními částmi algoritmu pro řešení puzzle. Nebo při každém určení hrany na hodnotu čára trasovat její oba koncové body a zjistit, zda netvoří cyklus, nebo se neliší o jednu hranu. Avšak ani jedno

z těchto řešení není ideální (z hlediska časové efektivity). Příkladem může být dlouhý souvislý segment čar, který budeme prodlužovat na některém z jeho konců.

Idea algoritmu

Budeme se zabývat nikoliv hranami, ale mřížovými body. Zavedeme nad nimi relaci ρ, kterou definujeme následovně:

Nechť a, b jsou mřížové body, potom $(a, b) \in \rho \Leftrightarrow$ existuje souvislá cesta z a do b po hranách označených čarou (definice je poněkud vágní, lze formulovat precizněji, avšak pro účely algoritmu je postačující a intuitivnější).

Předně si povšimněme, že relace ρ je ekvivalence (snadno lze dokázat reflexivitu, symetrii i tranzitivitu). Potom je ale faktorový rozklad množiny všech mřížových bodů podle ekvivalence ρ disjunktní. Dokážeme sporem. Nechť X a Y jsou dvě různé faktorové třídy (dle ekvivalence ρ) a nechť existuje prvek u z množiny mřížových bodů takový, že $u \in X$ a zároveň $u \in Y$. Potom pro každý prvek $z \in X$ takový, že $(z, u) \in \rho$, vyplývá, že $z \in Y$. Pomocí symetrie a tranzitivity nakonec dostaneme, že $X = Y$ a tudíž máme spor s tím, že X a Y byly různé faktorové třídy.

Realizace

Pro výše popsaný rozklad se nejlépe hodí struktura zvaná Disjoint-Find-Union (zkráceně DFU, viz článek [4]). Ve velmi rychlém čase (podrobnou analýzu naleznete taktéž v [4]) lze provádět dvě základní operace. První operací je find, která zjistí, zda dva prvky leží ve stejné podmnožině rozkladu a druhou je union zajišťující sjednocení dvou podmnožin do jedné. Při vlastní implementaci využívám obě možná urychlení – union by rank i path compression.

Navíc si pro každou podmnožinu pamatují koncové body (neboli dvojici mřížových bodů, jenž leží na okrajích souvislého úseku čar) a délku segmentu. Udržuji si ještě údaj o celkovém počtu segmentů, osahujících alespoň 2 mřížové body. To přesně odpovídá počtu souvislých segmentů čar v puzzle.

Posledním potřebným údajem je počet čar, jež chybí ke splnění všech nápověd. Tento počet je na začátku, před vlastním řešením puzzle, inicializován na součet všech nápověd v puzzle.

Popis algoritmu

Algoritmus pro kontrolu krátkých cyklů je spuštěn vždy v okamžiku, kdy je vydedukován stav hrany a jedná se o čáru (u křížku nemá smysl). Nejprve zjistíme mřížové body cp_1 a cp_2 u vydedukované hrany a zavoláme funkci union(cp_1, cp_2). Ta nám sjednotí dvě podmnožiny obsahující cp_1 a cp_2, nastaví nové koncové body sloučeného segmentu a upraví celkový počet segmentů. Rovněž se upraví údaj o počtu čar chybějících ke splnění všech nápověd podle dvou sousedních políček vydedukované hrany. V závislosti na pozici nových koncových bodů funkce union vrací některý z těchto stavů

a) žádná událost
b) chybí jediná hrana k vytvoření smyčky
c) segment se uzavřel, vznikla smyčka

Pokud nastane situace a, potom algoritmus končí. Pokud nastane poslední možnost, tedy že vznikla smyčka, pak provedeme následující test. Jestliže celkový počet segmentů je roven jedné a zároveň počet čar chybějících ke splnění všech nápověd je nulový, potom dočasně prohlásíme toto uskupení za řešení puzzle. Necháme doběhnout veškeré ostatní algoritmy pro řešení, a pokud nevznikl nikde spor, tak jde skutečně o jednoznačně dané řešení.

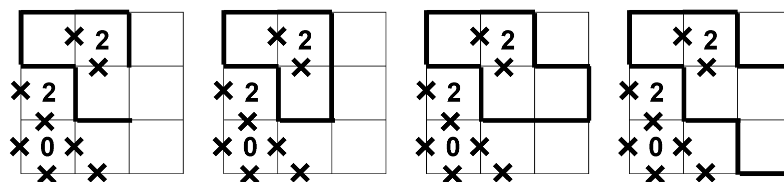
Pokud nastane druhá možnost, tak postupujeme následovně. Nejprve si určíme tzv. kritickou hranu, tj. ta hrana, která by nám vytvořila cyklus.

1) pokud je počet segmentů větší jak jedna, pak z toho vyplývá, že existuje alespoň ještě jeden segment. Nelze tedy nově vzniklý segment uzavřít a kritická hrana musí být křížek.
2) jinak máme pouze jediný segment

a) pokud je počet čar chybějících ke splnění všech nápověd větší než 2, pak z toho plyne, že taktéž nelze segment uzavřít. Zbyla by alespoň jedna nesplněná nápověda, a tedy kritickou hranu rovněž prohlásíme za křížek.
b) jinak se rozhodujeme dle sousedních políček kritické cesty. Pokud by i při jejích uvážení zbyla nějaká nesplněná nápověda, tak kritickou hranu můžeme prohlásit za křížek.

Důležité upozornění

V tomto algoritmu je velmi důležité, že pouze zakazujeme možnosti, které by nám s jistotou vytvořili krátký cyklus. Uvažme situaci, kdy nám chybí jediná hrana k uzavření segmentu čar. Navíc víme, že počet všech segmentů je roven jedné a máme splněn i počet čar chybějících ke splnění všech nápověd. Přesto nesmíme prohlásit toto uskupení za řešení. Řešení se musí vynutit nějakou jinou částí algoritmu pro řešení puzzle. Důvod proč to děláme právě takto, je zachování jednoznačného řešení puzzle. Pokud bychom tuto vlastnost nedodržovali při řešení puzzle, potom bychom nemohli ani generovat puzzle s jednoznačným řešením, jelikož při své práci používá algoritmy řešení. Situaci si objasníme na obrázku.



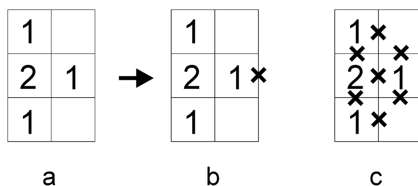
Obrázek 2.7.2 – nejednoznačnost puzzle

Na obrázku 2.7.2 zcela vlevo je vyobrazena situace popsána výše se všemi předpoklady splněnými. Přesto segment uzavřít nemůžeme – nemáme jistotu, zda jde skutečně o řešení, což dokládá trojice možných řešení.

2.8 3-ZÁMKY

Motivace

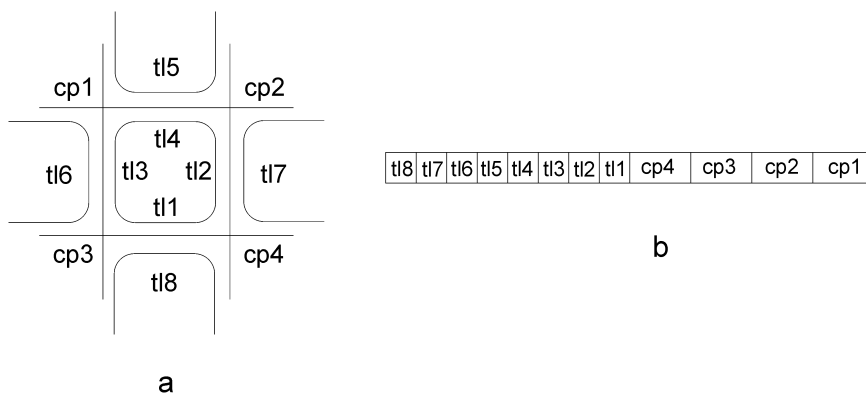
Představme si situaci znázorněnou na obrázku 2.8.1 v části a. Pomocí dosud popsaných metod nelze dospět k žádné hraně, přestože jedna hrana vydedukovat lze (viz část b). V části c je zobrazen spor, kterého bychom dosáhli, pokud by byla hrana nastavena na opačnou hodnotu.



Obrázek 2.8.1 – motivační příklad pro zavedení 3-zámků

Princip

Nový typ zámků, který zavedeme, bude oproti klasickému (2-zámku) zachycovat možný stav trojice sousedních hran. Vyjadřuje povolené počty čar ze tří hran, nad kterými je tento zámek vybudován. Například u nápovědy 2 jsou povolené počty čar u 3-zámků pouze 1 nebo 2.



Obrázek 2.8.2 – interakce zámků a jejich reprezentace

Troj-zámky budeme využívat pouze v souvislosti s políčky. Na obrázku 2.8.2 v části a je zachyceno rozmístění osmi troj-zámků. Takovéto rozmístění je vhodné z toho důvodu, že umožňuje interakci s okolím. Příkladem může být následující situace. Pokud během aplikace pravidel vydedukujeme změnu v zámku tl1, potom to ale zároveň znamená změnu zámku tl5 v políčku bezprostředně umístěným pod tímto políčkem.

Jeden troj-zámek zabírá 4 bity (možné počty jsou totiž 0, 1, 2, 3). Pravidla, využívající tento typ zámků, mají obdobný tvar, jako tomu je u klasických zámků.

Na obrázku 2.8.2 v části b je tento tvar zachycen. Potřebná velikost pro jedno pravidlo je 4 x 7 + 8 x 4 = 60 bitů, a tedy použijeme jedno 64bitové číslo.

Pro použití je zapotřebí pole, o velikosti celkového počtu políček, zachycující přípustné troj-zámky v jednotlivých políčkách.

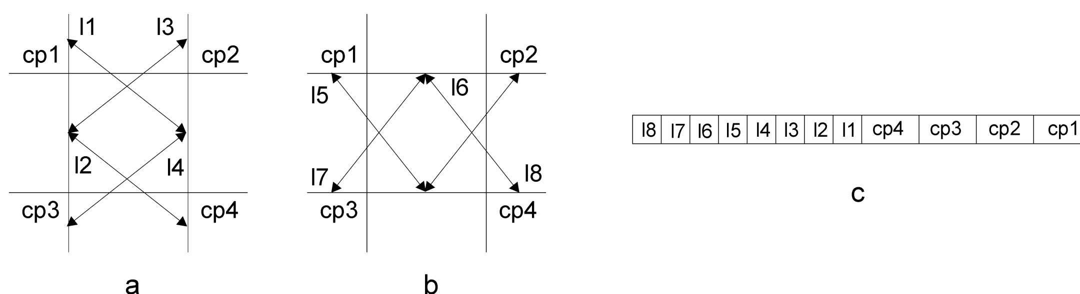
2.9 DIAGONÁLNÍ ZÁMKY

Motivace

Jedním z nejdůležitějších faktorů u pravidel je schopnost interakce – přenos informací do okolí. Navrhl jsem tzv. diagonální zámky, které toto také umožňují.

Princip

Obrázek 2.9 zachycuje strukturu diagonálních zámků. Pro zlepšení čitelnosti je struktura rozdělena do dvou částí (a a b), přičemž jejich složením dostaneme jedno pravidlo.



Obrázek 2.9 – struktura a reprezentace diagonálních zámků

Využíváme klasických CAL zámků, tedy zámků mezi dvěma hranami. Přenos probíhá jednoduchým způsobem. Nechť například je změněn zámek l5. Potom se tato změna projeví v políčku nalevo od tohoto a ovlivněný zámek tam bude l6.

Velikost jednoho pravidla je 4 x 7 + 8 x 3 = 52 bitů. Opět stačí 64bitové číslo. Způsob reprezentace je zachycen na obrázku 2.9 v části c.

Pro použití je zapotřebí pole, o velikosti celkového počtu políček, zachycující přípustné diagonální zámky v jednotlivých políčkách.

2.10 OBARVOVÁNÍ POLÍČEK

Motivace

K řešení puzzle lze s úspěchem také využívat vlastnosti topologie. Jde o jiný přístup než u pravidel založených na nápovědách a hranách a často značně pomáhá se řešením zejména obtížnějších puzzle. Princip vychází z následující věty.

Jordanova věta o kružnici – Každá topologická kružnice rozděluje rovinu na dvě souvislé oblasti.

Z této věty plyne, že rovina je topologickou kružnicí rozdělena na vnitřek a vnějšek. Dále si můžeme povšimnout toho, že libovolná přímka, která topologickou kružnici protíná, má právě sudý počet průniků (ve smyslu bodů). Toho lze využít k následujícímu jednoduchému pozorování.

Pozorování – Pokud existuje v puzzle řádek (sloupec), který má určeny všechny hrany až na jedinou, potom lze tuto zbývající hranu jednoznačně určit (počet čar musí být sudý).

Tím však možnosti topologie zdaleka nekončí. Dále si v textu popíšeme ideu obarvování políček.

Idea algoritmu

Jak již bylo řečeno, topologická kružnice nám rozdělí rovinu na vnitřek a vnějšek. Políčka mimo hrací plochu (po obvodu) si můžeme představit jako obarvená barvou značící vnějšek.

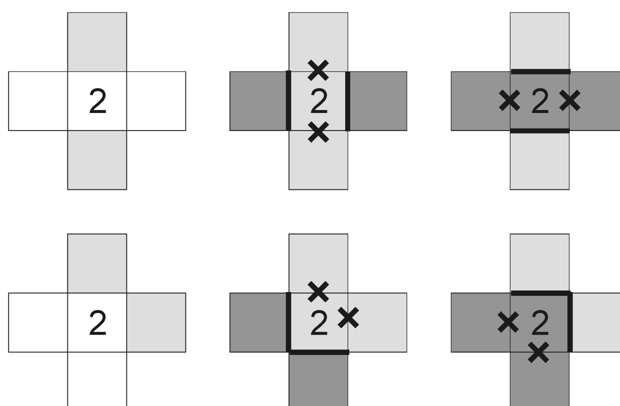
Základní pozorování

Pokud známe hodnotu hrany a barvu jednoho ze sousedních políček, potom lze jednoznačně určit barvu druhého sousedního políčka. Čára značí přechod z vnějšku do vnitřku (nebo naopak), a tudíž barva druhého políčka musí být opačná. Naopak křížek nám značí, že zůstáváme ve stejné souvislé oblasti (vnitřek / vnějšek), a barvy obou políček musejí být shodné. Toto pozorování lze využít i obráceně. Ze znalosti barev dvou sousedních políček lze určit hranu mezi nimi (viz obrázek 2.10.1)

Obrázek 2.10.1 – základní techniky obarvování

Po osvojení této metody lze vydedukovat barevná pravidla pro políčka s nápovědami. Budeme uvažovat čtveřici políček, jež sousedí s políčkem s nápovědou (tvoří kříž spolu s daným políčkem).

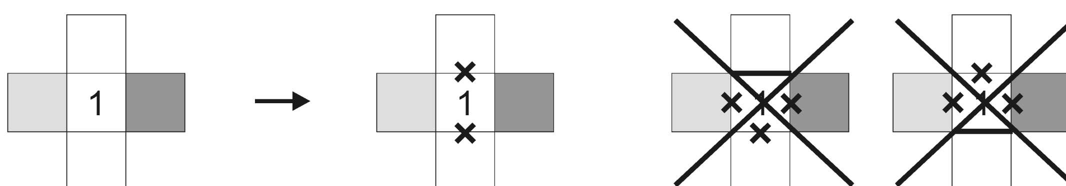
Za prvé lze dedukovat z barev na kříži zbylé barvy. Jako příklad uvedu následující. Pro políčko s nápovědou 2 plyne, že na kříži jsou 2 políčka s barvou vnějšku a 2 políčka s barvou vnitřku. Toto pravidlo lze využít několika různými způsoby v závislosti na tom, co víme o barvách na kříži (viz obrázek 2.10.2).



Obrázek 2.10.2 – odvození barevného pravidla pro nápovědu 2

Pro ostatní nápovědy lze odvodit pravidla podobným způsobem jako na obrázku 2.10.2.

Navíc lze odvozovat z barev na kříži hrany, případně naopak. Jako příklad nám poslouží následující obrázek (2.10.3).

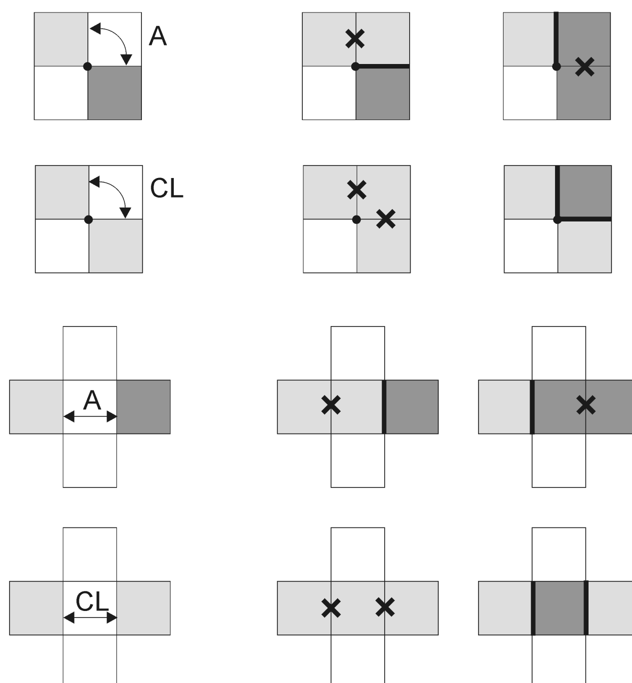


Obrázek 2.10.3 – odvození hran z barev na kříži

Vytvoření pravidel, která by brala v potaz zároveň barvy na kříži a hrany, by bylo vcelku komplikované (je zde velké množství různých konfigurací – počet určených barev, jejich pozice a barva, a k nim hrany na různých pozicích a v různých stavech (neurčená hrana, čára, křížek)). Přesto se nabízí elegantní a jednoduché řešení, na které jsem přišel.

Propojení barev a zámků

Myšlenka spočívá v tom, že se nebudeme vůbec zajímat o hrany. Pouze nám půjde o vztah mezi barvami a zámky (jak v mřížových bodech, tak v políčkách - příčné hrany). To nám jednak značně zjednoduší práci a navíc žádné informace neztratíme. Již dříve jsme si totiž popsali bijektivní dedukci hrany-zámky a nyní zavedeme taktéž bijektivní dedukci barvy-zámky. Složením těchto dedukcí dostaneme taktéž bijekci barvy-hrany. Na obrázku 2.10.4 jsou nastíněny veškeré potřebné dedukce mezi barvami a zámky spolu s jejich odvozením (stačí 2 pro mřížové body a 2 pro políčka).



Obrázek 2.10.4 – dedukce mezi barvami a zámky

Je zřejmé, že pravidlo lze zjednodušeně popsat tak, že pokud máme dvě políčka, která od sebe dělí jedno jediné políčko, potom pokud jsou barvy shodné, musí zde být zámek CL (crosses, lines), a pokud jsou opačné, potom zde musí být zámek A. Jak již jsem zmínil, jde o bijektivní deduktivní pravidlo, tudíž platí i naopak. Zámek CL implikuje shodné barvy, zámek A opačné barvy.

Realizace

V průběhu řešení puzzle budeme provádět následující úkony. Jednak to budou dedukce zámků z barev a barev ze zámků a to v políčkách i v mřížových bodech. Dále sjednocování barev či nastavení příznaku o tom, že barvy musejí být opačné.

Při realizaci nebudeme uvažovat pouze dvě barvy (vnitřek a vnějšek), jak tomu bylo doposud a jak je běžné při řešení puzzle na papíře či počítači, nýbrž pro získání co nejlepších výsledků použijeme následující ideu. Na počátku budou mít veškerá políčka svou vlastní barvu (navzájem rozdílnou). Políčka mimo hrací

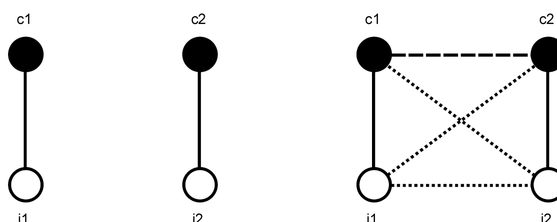
plochu se při inicializaci sloučí a budou mít barvu vnějšku. V průběhu algoritmu řešení lze tedy slučovat či oddělovat i takové barvy, u kterých není doposud známo, zda budou vnějškem nebo vnitřkem. Tím docílíme více dedukcí.

Nejprve si ukážeme, jak probíhá slučování a oddělování barevných komponent (ve smyslu dedukcí), a poté způsob, jakým lze efektivně uchovávat informace o barvách, jejích vztazích a jak na této struktuře provádět operace slučování a oddělování taktéž efektivním způsobem.

Slučování a oddělování barevných komponent

V průběhu algoritmu platí invariant, že komponenty barev mají z hlediska inkluze maximální možnou velikost. Tedy nemůže se stát, že by dvě políčka měla stejnou barvu a nebyla by ve stejné komponentě. Toto triviálně platí na počátku (každé políčko má odlišnou barvu) a ukážeme si, že to platí po celou dobu běhu algoritmu.

Ke každé barevné komponentě může, ale nemusí, existovat komponenta opačné barvy. Bez újmy na obecnosti předpokládejme, že ji má každá barva. Pokud by ji neměla, tak si opačnou barvu můžeme představit jako prázdnou množinu. Mějme barvy c_1 (s opačnou barvou i_1) a c_2 (s opačnou barvou i_2). Nejprve uvažme, že bychom chtěli barvy c_1 a c_2 sloučit. Musíme provést následující kroky (schéma je zachyceno na obrázku 2.10.5)



Obrázek 2.10.5 – sjednocení dvou barevných komponent

Prvním krokem je vlastní sjednocení barev c_1 a c_2. Při něm musíme kvůli dedukcím také projít políčka jedné z těchto barev (např. c_1) a zkoumat, zda nesousedí s druhou barvou (c_2). Pokud ano, potom lze hranu mezi těmito políčky označit křížkem (c_1 a c_2 budou tvořit jednu komponentu). Dále je potřeba sjednotit podobným způsobem barvy i1 a i2 a případné hrany mezi nimi označit křížky.

Tímto dedukce nekončí – povšimněme si, že vztah - barva c_1 je opačná s barvou i_2 - je pro nás nový. To samé platí i pro barvy c_2 a i_1. Opět je zapotřebí projít políčka, a pokud spolu tyto barvy sousedí, tak je oddělíme čarou.

Nechť sjednocením barev c_1 a c_2 dostaneme barvu c, a sjednocením barev i1 a i_2 dostaneme barvu i. Potom barvě c musíme nastavit příznak, že barva i je opačná (a zároveň i naopak – jelikož relace opačných barev je symetrická). Tímto krokem pro nás sjednocování končí.

Proces oddělení barevných komponent (nastavení příznaku, že jsou opačné) je velmi podobný. Stačí použít výše zmíněné schéma pomocí jednoduchého triku - je zapotřebí prohodit c_2 s i_2. Poté je dedukce popsaná výše zcela shodná.

Struktura pro uchovávání barevných komponent

Snadno si lze povšimnout, že při práci algoritmu se barevné komponenty pouze zvětšují (ve smyslu inkluze). Mohlo by se tedy zdát, že je na místě opět DFU jako tomu bylo u jednoduché kontroly krátkých cyklů, ale opak je pravdou. Při dedukcích potřebujeme celou komponentu projít, což při běžné implementaci DFU nelze (důvodem je jeho struktura – strom, avšak s opačně orientovanými hranami, směrem ke kořeni).

Nevhodné pro reprezentaci komponent jsou také různé seznamy (List), jelikož jakákoliv komponenta může mít velikost až O(|N|), kde N je počet políček, a museli bychom tedy často tyto seznamy realokovat (a s tím je spojena i časová režie na provedení), nebo naddimenzovat všechny tyto seznamy (což vede k plýtvání pamětí).

Pro udržení potřebných informací o všech komponentách zavedeme 5 polí o velikostech N.

- colors – udává barvu daného políčka
- opposite_colors – uchovává barvu opačnou ke komponentě s daným číslem
- first_cell – pro danou barevnou komponentu udává první políčko, které obsahuje (políčka v komponentě jsou uspořádána dle jejich očíslování)
- next_cell – obsahuje číslo dalšího políčka, které je ve stejné barevné komponentě jako aktuální políčko (nebo příznak, že další políčko této barvy již není)
- col_sizes – udává velikosti barevných komponent

Využitím této struktury nedochází k plýtvání s pamětí, jelikož po celou dobu algoritmu platí, že součet velikostí jednotlivých komponent je roven N. Jednotlivé komponenty jsou reprezentovány pomocí ukazatelů first_cell a next_cell (jakoby spojové seznamy). Tím máme zajištěno maximální využití struktury a tím pádem i paměti.

Díky ukazatelům by nebylo zapotřebí pole, které nám udává velikost barevných komponent, ale přesto má své opodstatnění. Zajišťuje nám efektivní chod algoritmu. Pro účel popisu si představme zjednodušenou situaci, kdy barvy nemají k sobě opačnou barvu. Dejme tomu, že máme dvě barvy, které chceme sjednotit, přičemž jedna obsahuje pouze jediné políčko a druhá je obrovská (např. N/2). Jak bylo popsáno výše, při sjednocení dvou barev se snažíme provádět dedukce na základě sousednosti políček. Pokud dvě políčka, každé od jedné z těchto barev, spolu navzájem sousedí, potom mezi nimi musí být křížek. Pokud bychom kontrolovali první barvu, pak počet kontrol je roven čtyřem (jediné políčko a má 4 sousedy). Pokud bychom kontrolovali druhou barvu, pak počet kontrol bude 2N. Rozdíl v počtu kontrol je tedy výrazný a daleko efektivnější bude kontrolovat první políčko.

Popis algoritmu

Algoritmus sjednocování (oddělování) komponent probíhá následovně. Nechť sjednocujeme barvy c_1 a c_2. Nejprve určíme opačné barvy i_1 a i_2. Podle součtu $c_1 + i_1$ a součtu $c_2 + i_2$ se budeme rozhodovat, zda přebarvíme barvu c_2 na barvu c_1 či naopak. Kvůli efektivitě budeme přebarvovat tu, která má menší součet.

Jelikož po celou dobu platí, že políčka v jednotlivých komponentách jsou vzestupně uspořádána dle svého pořadového čísla, musíme procházet obě komponenty zároveň. Větší komponentě z těchto dvou upravujeme záznamy first_cell a next_cell. Nad tou menší provádíme dedukce (ty jsou časově náročnější) a jejím políčkům přepisujeme záznamy v poli colors na hodnotu větší komponenty. V průběhu algoritmu také upravujeme i_1 a i_2 a dedukce mezi nimi a barvami c_1 a c_2 – vše se odvíjí od existence i_1 a i_2. Rovněž pak musíme nastavit výslednou velikost nové barevné komponenty a vztah o tom, že barvy $c = c_1 \cup c_2$ a $i = i_1 \cup i_2$ jsou navzájem opačné.

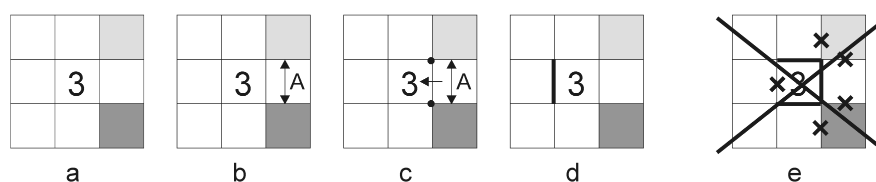
Postupem času jsem dospěl k tomu, že je ještě zapotřebí přidat sousedy všech políček, která kontrolujeme kvůli dedukcím a přepisujeme jejich barvu, do fronty políček. Důvod je ten, že častokrát lze provádět barevné dedukce na kříži okolo těchto políček, jelikož se barva jednoho z políček na kříži změnila (a může být shodná / opačná s nějakým jiným políčkem na kříži, což nám implikuje zisk zámku).

Celý princip využívající obarvování probíhá následovně. Kdykoliv je určena nějaká hrana, potom barvy sousedních políček sjednotíme či rozdělíme v závislosti na hodnotě nově určené hrany (křížek nebo čára). Při sjednocování či rozdělování barev naopak dedukujeme hrany. Dále při kontrole políčka nebo mřížového bodu vyvozujeme zámky z okolních barev. A rovněž i naopak – vyvozujeme okolní barvy ze zámků.

Během průběhu algoritmu pro řešení puzzle je mnohokrát volána metoda pro sjednocování (rozdělování). Přesto celková složitost algoritmu v rámci řešení je jen $O(|N|^2)$.

Praktická ukázka

Předvedeme si příklad dedukcí barev políček. Předpokládejme situaci znázorněnou na obrázku 2.10.6 (část a).



Obrázek 2.10.6 – příklad barevných dedukcí

K této situaci jsme dospěli v průběhu chodu algoritmu pro řešení s využitím slučováním a oddělováním barevných komponent. Na obrázku 2.10.6 (část b) získáme díky barevným dedukcím v políčku ve druhém řádku vpravo zámek A. Tím se signalizuje změna stavu v políčku a spustí se zde dedukce pomocí pravidel. Pravidla nám vyloučí některé možné konfigurace a tím ovlivní i stavy svých mřížových bodů (obrázek 2.10.6 část c). Díky této změně se zavolá dedukce podle pravidel v políčku s nápovědou 3. Na dalším části obrázku 2.10.6 (část d) lze vidět jednu vydedukovanou hranu. Proč tomu tak je ukazuje poslední část obrázku 2.10.6 pomocí sporu (část e).

2.11 OBARVOVÁNÍ HRAN

Motivace

Obarvování hran je prostředek, který umožňuje interakci zámků v rámci širšího okolí díky přenosu informací přes hrany a jejich vzájemným vztahům, nikoliv jen v lokálním pojetí. To umožňuje další dedukce a tedy i vyřešení obtížnějších puzzle. Tato technika byla náznakem zmíněna v programu LoopDeLoop [5], nikoliv však popsána.

Má implementace se navíc snaží vytěžit co nejvíce ze vzájemné interakce zámků (netriviální dedukce) a jako hlavní motivaci pro zavedení obarvování hran pro mě bylo zavedení pokročilé kontroly krátkých cyklů nad obarvováním hran. Tato technika, myslím si, ještě nebyla nikde použita a má velmi časté použití (viz následující kapitola).

Idea algoritmu

Základní idea je obdobná, jako je tomu u obarvování políček. Budeme uchovávat barevné komponenty a informace o opačných barvách. Dedukce pak budou opět dvojího druhu – jednak dedukce barev hran ze zámků a za druhé dedukce zámků z barev hran.

Realizace

Informace potřebné pro obarvování hran uchováváme ve třídě a pro práci algoritmů využíváme následujících polí.

- *colors* barvy jednotlivých hran
- *opposite_colors* číslo opačné barvy k dané barvě
- *first* ukazatel na první hranu dané barevné komponenty
- *next* ukazatel na následující hranu v dané barevné komponentě
- *order_in_group* pořadí hrany v barevné komponentě
- *color_size* velikost barevné komponenty
- *active* příznak, zda je daná barva stále aktivní

Barva se může stát neaktivní, a to právě ve chvíli, kdy nějaká hrana v určité barevné komponentě změní stav z neurčené hrany na křížek/čára. Od této chvíle nemá smysl se již nadále touto komponentou zabývat, jelikož ostatní hrany mohou být určeny (za pomocí zámků) jednoduššími algoritmy pro řešení puzzle. Pořadí hrany v barevné komponentě je zde připraveno pro použití při detekci krátkých cyklů – v základní verzi algoritmu nemá uplatnění.

Idea je následující. Pokud mezi dvěma hranami se změní stav zámku, tak potom pokud jde o zámek CL, sloučíme barvy reprezentované těmito hranami. V případě zámku A barvy oddělíme.

Algoritmy pro sjednocení dvou barevných komponent jsou téměř identické s těmi, které mají na starost udělat dvě barvy opačnými. Proto si zde popíšeme pouze první variantu – sjednocení dvou barev.

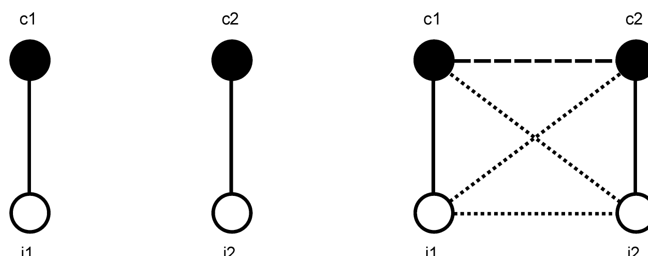
Algoritmus

Popíšeme si algoritmus pro sjednocení dvou hran e_1, e_2 do jedné barevné komponenty.

Označme:

- c_1 barevná komponenta, do které patří hrana e_1
- c_2 barevná komponenta, do které patří hrana e_2
- i_1 barevná komponenta opačná k c1
- i_2 barevná komponenta opačná k c2

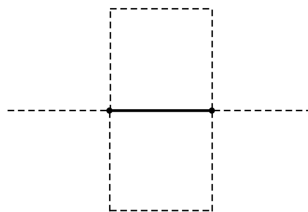
Nejprve určíme komponenty c_1 a c_2. Pokud je alespoň jedna komponenta neaktivní, tak zde končíme (nemá smysl tuto akci provádět). Dále otestujme, zda barvy c_1 a c_2 nejsou shodné, pokud by byly, tak rovněž nemá smysl pokračovat (hrany e_1 a e_2 již jsou v jedné komponentě). Posledním z testů je, zda nejsou barvy c_1 a c_2 navzájem opačné. V tom případě končíme s tím, že se jedná o chybu v puzzle. Nelze sloučit navzájem opačné komponenty. Opačné barvy i_1 a i_2 mohou existovat, ale samozřejmě nemusejí.



Obrázek 2.11.1 – dedukce při sjednocení barev

Připomeňme si opět obrázek 2.11.1 pro dedukování nad barvami. Tím, že sjednotíme barvy c_1 a c_2, můžeme provést další dedukce ze znalosti toho, že se sjednotí rovněž barvy i_1 a i_2 a barvy c_1 s i_2 se stanou opačnými rovněž jako c_2 s i_1.

Podstatná je při slučování komponent zpětná vazba – dedukce zámků ze znalosti barev. Při sjednocování komponent, projdeme hrany všech barev – c_1, c_2, i_1, i_2. Ke každé z těchto hran určíme 8 bezprostředních sousedů (viz obrázek 2.11.2).



Obrázek 2.11.2 – bezprostřední sousedé hrany

Vezměme tedy jednu hranu z těchto 8 sousedů. Určíme její barvu a CAL zámek s původní hranou. Při práci algoritmu používáme ještě speciální haldu, do které ukládáme tyto informace o sousedech (spolu s informací, ke které z komponent – c_1, c_2, i_1, i_2 – náleží). Klíčem pro tuto haldu je barva vkládané hrany.

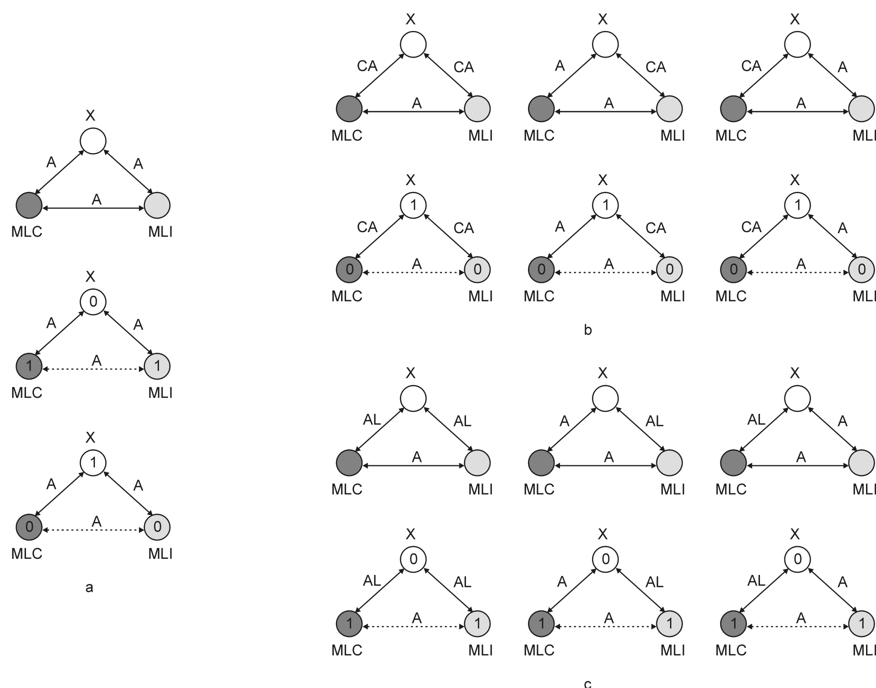
Po naplnění haldy začnou vlastní dedukce. Předpokládejme, že barva c bude sjednocením barev c_1, c_2 a barva i bude sjednocením barev i_1, i_2, a zároveň také barvou opačnou k barvě c. Zavedeme ještě dvě proměnné následovně.

- MLC – minimální zámek určité barvy k barvě c (tj. vzhledem k barvám c_1, c_2)
- MLI – minimální zámek určité barvy k barvě i (tj. vzhledem k barvám i_1, i_2)

Vyzvedneme jednu barvu z haldy, poté zjistíme, jaké komponentě byla sousedem, a jaký s ní má zámek. Z těchto informací inicializujeme hodnoty MLC a MLI. Z haldy poté vyzvedneme další barvu. Víme, že bravy jsou uspořádané vzestupně (díky vlastnostem min-haldy). Mohou nyní nastat dvě možnosti. První je ta, že vyzvednutá barva se liší od minulé (má vyšší číslo). Z hlediska dedukcí pro nás tato možnost nemá význam. V případě, že barva je shodná s předchozí, uděláme průnik hodnoty jejího zámku s patřičnou proměnnou (buď MLC nebo MLI v závislosti na tom, které komponentě byla tato hrana sousedem). Takto pokračujeme, dokud z haldy vyzvedáváme stále stejnou barvu. Pokud již v haldě daná barva není, projdeme znovu všechny vyzvednuté hrany s touto barvou. Testujeme, zda se u hrany zámek shoduje s minimální hodnotou zámku (MLC/MLI). Pokud ne, pak na tomto místě máme vydedukovanou změnu zámku. Musíme tedy upravit stav zámku mřížového bodu či políčka a vložit jej do příslušné fronty (pro další dedukce).

Může se však stát, že vydedukujeme rovnou, jakou hodnotu musí mít barevná komponenta ve smyslu křížek/čára. Stane se tak pokud nějakému z minimálních zámků zbude jako jediný kandidát C nebo L.

V závislosti na dvojici minimálních zámků MLC a MLI lze provést taktéž dedukce, jak ukazuje obrázek 2.11.3.



Obrázek 2.11.3 – dedukce mezi barvou s užití MLC a MLI

V části a na obrázku 2.11.3 je znázorněna situace, která ukazuje na chybné puzzle, tj. stav kdy MLC = A a zároveň MLI = A. Důkaz je proveden rozborem možností – hodnoty křížek (nula) a čára (jedna), čárkovaně je pak naznačen vyniklý spor. Na obrázku 2.11.3 v části b je naznačena další situace. V případě, že MLC = A/AC a zároveň MLI = A/AC, potom vyzvednutá barva musí být nastavena na hodnotu křížek. Pokud by měla hodnotu čára, vede to ke sporu, jak je naznačeno na obrázku 2.11.3. Analogickou situaci zachycuje obrázek 2.11.3 v části c, ze kterého implikuje, kdykoliv je MLC = A/AL a zároveň MLI = A/AL, potom barva musí mít hodnotu čára.

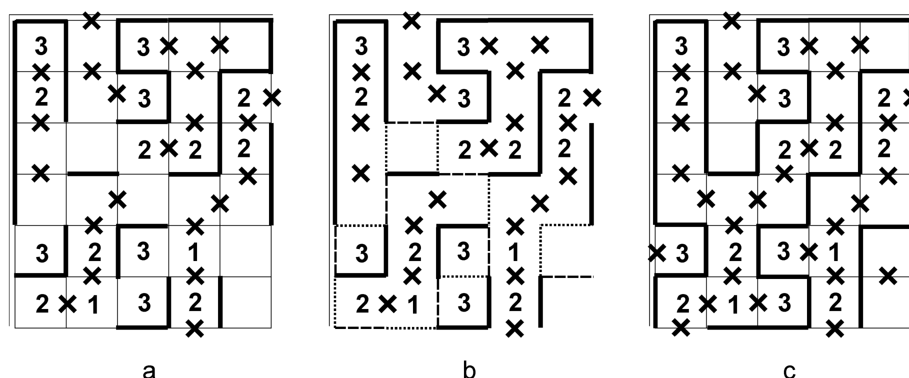
Sjednocování barevných komponent lze provádět i nad dvojicemi diagonálních hran díky znalostem zámků mezi nimi, pokud používáme diagonální zámky.

2.12 POKROČILÉ DEDUKCE KRÁTKÝCH CYKLŮ NAD OBARVENÍM HRAN

Motivace

V průběhu řešení puzzle nastává často situace, kdy lze vydedukovat hranu díky tomu, že se zde vyskytuje pro uživatele (hráče) zřejmý krátký cyklus. Dříve

popsaná pravidla si s takovýmito stavy nedokážou poradit, přestože se nejedná o složitou situaci (z pohledu hráče). Metodu si nastíníme na následujícím obrázku.



Obrázek 2.12 – dedukce krátkých cyklů nad obarvením hran

Pokud se podíváme na výřez puzzle (jde o levý horní roh), uvidíme herní situaci, se kterou si pravidla již dále nedokážou poradit (obrázek 2.12, část a). Za povšimnutí stojí to, že většina dvojic sousedních neurčených hran sdílí mezi sebou zámek A (Antilocked). Jinými slovy musejí mít navzájem opačnou barvu, co se týče obarvování hran. Pomocí techniky slučování a oddělování barevných komponent hran, dostaneme situaci znázorněnou na obrázku 2.12 v části b, kde pro větší přehlednost byla vynechána mřížka vodících čar. Zde je vyznačena čárkovaně jedna barevná komponenta a druhá (k ní opačná) je naznačena tečkovaně. Pokud se více zaměříme na čárkované hrany, uvidíme, že společně se segmenty čar vytvářejí krátký cyklus. Přesněji řečeno rovnou dva. Z toho plyne, že čárkované hrany musejí mít hodnotu křížek (jinak by vznikly cykly). Rovněž z toho nepřímo plyne, že tečkované hrany musejí být čarou, jelikož jde o opačnou barevnou komponentu vzhledem k čárkované. Znázornění po dedukcích vidíme na obrázku 2.12 v části c.

Princip algoritmu

Zde využijeme pole, které nám určovalo pořadí hrany v dané barevné komponentě. Slouží k urychlení práce algoritmu a k jeho zjednodušení. Nejprve ještě zavedeme nové pole o velikosti kontrolované barevné komponenty, které nám bude udávat, zda hrana s jistým pořadovým číslem již byla kontrolována na cyklus (opět slouží ke zrychlení práce).

Je nutné si uvědomit, že jedna barevná komponenta hran nemusí být souvislá. Obecně se jedná o několik segmentů. Další důležitý poznatek je to, že jedna barva může tvořit i více krátkých cyklů (viz příklad v motivační části).

Kontrola tedy probíhá následovně. V cyklu zkoumáme hrany dle jejich pořadí. Testujeme je na cyklus pouze v případě, pokud má v poli nastavenu hodnotu, že dosud nebyla testována na cyklus. Testování pak probíhá tak, že hranu nejprve označíme za již testovanou a určíme její dva mřížové body. Jeden prohlásíme za začátek a druhý za konec. Ze začátku spustíme prohledávání. Hledáme souvislé pokračování ve smyslu cyklu. To může být buď taková hrana, která má stejné obarvení nebo segment čar. V případě hrany se stejným obarvením

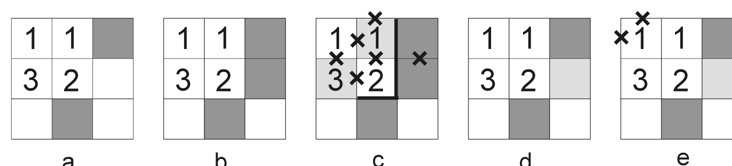
ji také označíme za již testovanou a přesuneme se do jejího druhého koncového (mřížového) bodu, kde pokračujeme obdobně. Pokud jde o segment čar, pak díky naší implementaci jednoduše zjistíme jeho druhý koncový bod, aniž bychom tento segment museli trasovat. Je to díky tomu, že máme pro každý souvislý segment čar uchovány a průběžně aktualizovány hodnoty jeho obou koncových bodů. Hledání souvislého pokračování může skončit ze dvou příčin. První je fakt, že jsme dorazili do konce původní testované hrany. V tom případě máme cyklus, dále se touto barevnou komponentou nezabýváme a veškeré hrany obsažené v této komponentě označíme křížkem. Druhou možnou příčinou ukončení prohledávání je to, že již nemáme kam pokračovat. Jinými slovy tato souvislá část barevné komponenty hran cyklus tvořit nemůže. Abychom předešli opakování tohoto samého prozkoumávání, budeme trasovat souvislé barevné pokračování ještě z druhé strany původní hrany. Veškeré hrany po této cestě označíme za již testované. Je zřejmé, že ty již také nemohou tvořit cyklus.

Otázkou je kdy a jaké barvy máme na cyklus testovat. To je velmi podstatné z hlediska celkové efektivity algoritmu. Pravidlo zní následovně. Budeme si v průběhu algoritmu řešení puzzle udržovat další frontu, do které budeme vkládat číslo barvy, pokaždé když dojde ke sloučení nějakých dvou barvených komponent (vkládáme tedy výslednou barvu). Barvu taktéž vkládáme do fronty, kdykoliv se rozšíří nějaký segment čar. Přesněji do fronty vkládáme barvy neurčených hran, které bezprostředně sousedí s koncovými body rozšířeného segmentu. Toto jsou jediné dvě události, které nám mohou podnítit vznik barevného krátkého cyklu. Barvy ke kontrole vybíráme z fronty pouze v případě, že již nelze nic jiného vydedukovat pomocí jednodušších metod (pravidel). Pokud těmito dedukcemi zjistíme, že by došlo ke krátkému cyklu, a tedy vydedukujeme u některých hran novou hodnotu (křížek), potom opět dáváme slovo jednodušším (a rychlejším) metodám pro řešení.

2.13 POKROČILÉ BAREVNÉ DEDUKCE V POLÍČKU

Motivace

Princip této metody objasní příklad. Jedná se pouze o část puzzle, které jsem se snažil ručně dořešit poté, co všechny dříve zmíněné metody nezaznamenaly žádný pokrok.



Obrázek 2.13.1 – motivace zavedení pokročilých barvených dedukcí v políčku

Na obrázku 2.13.1 v části a je zobrazena výchozí situace. Pokud bychom políčko v druhé řádce vpravo obarvili tmavou barvou (obrázek 2.13.1 část b), poté

by z toho implikovaly dedukce podobné části c na obrázku 2.13.1. Avšak to je nekonzistentní stav – všimněme si políčka s nápovědou 3, které smí obsahovat maximálně jeden křížek. Z toho vyplývá, že políčko musí mít světlou barvu (opačnou k tmavé), což je znázorněno na obrázku 2.13.1 v části d. Po aplikaci pravidel obdržíme v tomto výřezu puzzle ještě 2 křížky (část e na obrázku 2.13.1).

Idea algoritmu

Opět budeme používat pravidla. Avšak tato kontrola bude časově náročnější, než byly předchozí kontroly. Jak již bylo řečeno v kapitole o obarvování políček, při práci s barvami nepracujeme pouze se dvěma barevnými komponentami (vnitřek a vnějšek), nýbrž s více barevnými komponentami. Zavedeme proto další typ zámků – ten bude moci nabývat pouze dvou stavů S a O. S bude značit, že barvy jsou ve stejné barevné komponentě (Same) a O, že jsou v opačné (Opposite). Samozřejmě, že mezi dvěma barvami nemusí být žádný vztah. Zámek je složen ze dvou bitů (S a O). Na počátku nabývá hodnoty S = 1 a O = 1, což znamená, že vztah není dán – barvy mohou být buďto stejné nebo opačné. Jakmile je jeden z bitů nulový, potom z toho plyne vztah mezi barvami (ten, který zůstal nenulový). Nula v obou bitech by znamenala chybné puzzle.

Budeme potřebovat vztah mezi každou dvojicí z celkových 9 políček tak, jak je znázorněno na obrázku 2.13.2. V první části je naznačeno očíslování políček pro potřeby algoritmu, ve druhé části jsou potom popsány všechny možné vztahy mezi políčky (znázornění vztahů šipkami by bylo nepřehledné). Vztah - být ve stejné / opačné barevné komponentě - je symetrický, proto není zapotřebí více vztahů. Jen pro upřesnění dedukce se provádí nad políčkem s číslem 4, ostatní políčka jsou okolní.

0	1	2
3	4	5
6	7	8

0-1	1-2	2-3	3-4	4-5	5-6	6-7	7-8
0-2	1-3	2-4	3-5	4-6	5-7	6-8	
0-3	1-4	2-5	3-6	4-7	5-8		
0-4	1-5	2-6	3-7	4-8			
0-5	1-6	2-7	3-8				
0-6	1-7	2-8					
0-7	1-8						
0-8							

Obrázek 2.13.2 – popis vztahů mezi políčky pro účely pravidel

Pokud si chceme spočítat potřebnou velikost pravidla, musíme uvážit 4 stavy mřížových bodů a 36 vztahů mezi barvami. Výsledkem je 4 x 7 + 36 x 2 = 100 bitů. Pro realizaci budeme potřebovat dvě 64bitová čísla.

Realizace

Jedno pravidlo si rozdělíme do dvou částí. Potom při aplikaci pravidel je pravidlo splněno právě tehdy, když jsou splněny obě části zároveň. Také je zapotřebí složitější preprocessing, což je zapříčiněno velkým počtem dvojic políček (konkrétně 36), která musíme porovnat, zda mají stejnou či opačnou barvu. Pokud vydedukujeme nějakou změnu po kontrole všech pravidel, tak opět musíme projít tyto dvojice, abychom mohli sjednotit či oddělit barevné komponenty.

I přes tyto menší komplikace stále platí, že časová složitost tohoto algoritmu (běhu na jednom políčku) je konstantní pro nějakou dostatečně velkou konstantu (zjednodušeně C = 2 * 160 + 2 * 36). Není tedy závislá na velikosti puzzle.

2.14 POKROČILÉ BAREVNÉ DEDUKCE NAD HRANAMI V POLÍČKU

Motivace

Jde o možnost zachytit vztahy i mezi hranami, které spolu přímo nesousedí. Takovéto zámky nelze totiž propagovat dále. Přesto lze vyvozovat dedukce pomocí pravidel v závislosti na tom, zda hrany musejí být stejné (jsou ve stejné barevné komponentě), nebo opačné.

Idea algoritmu

Pokud uvážíme políčko a 12 sousedních hran okolo něj, povšimneme si, že spoustu zámků mezi dvojicemi hran již máme určenou (hrany sdílející stejný mřížový bod či diagonální zámky). Budeme uvažovat tedy pouze takové dvojice hran, mezi kterými ještě žádný vztah nebyl. Stejně tak, jako tomu bylo u pokročilých dedukcí nad obarvením políček, budeme pracovat s S/O zámky.

V první fázi zjišťujeme přípustné možnosti 4 mřížových bodů a jednotlivé vztahy mezi barvami u dvojic hran. Poté aplikujeme pravidla obvyklým způsobem a nakonec zpětně zkoumáme, zda se nějaké části nezměnily. V kladném případě tuto změnu nastavíme. Pokud jde o mřížový bod, upravíme jeho hodnotu, pokud jde o S/O zámek, pak sloučíme či rozdělíme barevné komponenty hran.

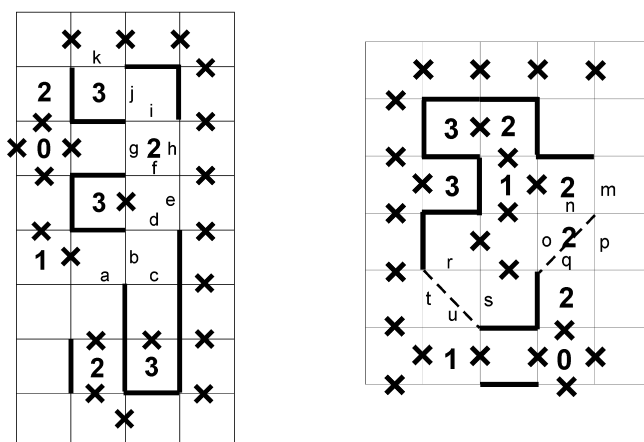
Realizace

Počet dvojic hran, mezi kterými jsme dosud neměly veden žádný vztah, je 26. Potom potřebná velikost pravidla je 26 x 2 + 4 x 7 = 80 bitů. Použijeme pro reprezentaci dvě čísla, jedno 64bitové a jedno 32bitové (přestože by postačovalo 16bitové).

2.15 POKROČILÉ DEDUKCE CEST

Motivace

V některých situacích dedukce na základě obarvení hran nestačí k zabránění zřejmých krátkých cyklů. Pod pojmem zřejmý mám na mysli očividných pro uživatele. Jde převážně o stav, kdy jedna hrana může vynutit vznik krátkého cyklu. Situaci si objasníme na následujících obrázcích (upozorňuji, že se jedná pouze o části puzzle).



Obrázek 2.15.1 – dedukce krátkých cyklů

Na prvním obrázku (2.15.1 vlevo) je vidět, že pokud zvolíme hranu b (nebo hranu g nebo i) a nastavíme její hodnotu na čáru, potom obdržíme krátký cyklus. Uvažme např. b jako čáru, potom z toho vyplývá, že a, c, d jsou křížky. Z toho plyne, že e musí být čára. Abychom nedostali cyklus, tak f musí být křížek. Z tohoto opět dostaneme, že h a g musejí být čáry. Dále i a j musejí být křížky a konečně hrana k bude čára. Takto ale obdržíme krátký cyklus a tedy hrana b nesmí být čára. Pro většinu uživatelů (hráčů) je tato situace zřejmá na první pohled. Ještě snadněji bychom obdrželi cyklus v případě, že zvolíme jako čáru hranu i (přesněji řečeno obdržíme hned dva krátké cykly).

U druhé části obrázku (2.15.1 vpravo) jde o poněkud jiný příklad, kde lze jednoduše vidět krátký cyklus, pokud bychom zvolili hraně m hodnotu čára. Vycházíme z obarvování hran (jež vychází ze zámků). Zřejmě hrany n, o musejí mít stejnou hodnotu. Podobně hrany p, q musejí mít stejnou hodnotu, avšak opačnou k hranám n, o. Tedy dostaneme jakési umělé propojení (bude definováno později), v obrázku naznačené přerušovanou čarou. Obdobně s hranami r, s, t, u. Jednoduše lze tedy usoudit, že hodnota hrany m musí být křížek.

Idea algoritmu

Zjednodušeně lze popsat algoritmus následovně. Vezmeme jeden souvislý segment čar a podíváme se na jeho koncové body. Nyní postupně zkoušíme možná pokračování koncových bodů a provádíme dedukce. Pokud dojdeme ke sporu lze takovéto pokračování zavrhnout.

Realizace

K práci algoritmu potřebujeme zásobník obarvení hran a sám algoritmus vychází z principu obarvování hran. Zásobník má omezenou velikost (20 barev) a obsahuje údaje o tom, jaké barvy hran byly určeny a na jakou hodnotu (křížek či čára). Díky této struktuře nemusíme měnit hodnoty hran, ale lze dedukovat pouze nad strukturou obarvování hran.

Popis algoritmu

1. Preprocessing – určení koncových bodů ep_1 a ep_2
2. Dedukce na základě ep_1 a ep_2
3. Zkoušení možných pokračování ep_1
4. Zkoušení možných pokračování ep_2

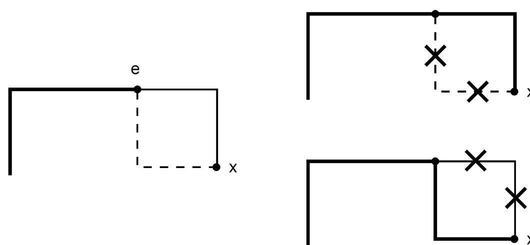
Preprocesing

Algoritmus dostane jako parametr číslo reprezentanta segmentu, který se má zkontrolovat. Nejprve otestujeme, zda jde stále o reprezentanta, což v podstatě znamená, zda jde o kořen v DFU stromu (viz kapitola Jednoduchá kontrola krátkých cyklů). Pokud kořenem není, tak se jím nezabýváme a končíme.

Zjistíme koncové body segmentu ep_1 a ep_2 (jde o čísla mřížových bodů, v nichž segment končí) a jeho délku – vše v konstantním čase. Dále je potřeba určit povolené směry, kterými by se segment mohl rozšířit – v tomto základním případě stačí zakázat jediný směr a to ten, ve kterém se nalézá čára (tudy pokračovat nemůžeme).

Pozorování (UP – umělé prodloužení)

Nechť e je koncový bod segmentu čar. Nazvěme barevným pokračováním koncového bodu e nejdelší souvislou cestu z bodu e po jediné barvě. Pokud pro každý z povolených směrů segmentu v koncovém bodě e platí, že barevné pokračování e vyúsťuje do stále stejného bodu x, potom lze původní segment rozšířit (pouze pro účely dedukcí cest) nahrazením bodu e za nový koncový bod x. Navíc platí, že pro x jsou povolené směry pro rozšíření právě ty, které neleží na žádném z barevných pokračování z původního koncového bodu e.



Obrázek 2.15.2 - umělé pokračování

Důkaz: Z obrázku 2.15.2 je pozorování zřejmé. Dodatek pozorování pak vyplývá ze základní vlastnosti mřížových bodů (princip sudosti).

Preprocessing potom spočívá v tom, že se snažíme získat co nejdelší segment čar z původně zadaného segmentu. Používáme k tomu funkci pro hledání pokračování, která bude popsána později při vlastní práci algoritmu. Zejména se ale v této fázi používá pozorování UP.

Poznámka: Již v této fázi si ale musíme dát pozor na to, zda nově získané koncové body ep_1 a ep_2 nejsou shodné – v takovém případě bychom se nechtěně zacyklili. Z tohoto žádné dedukce neplynou, pouze jsme se dozvěděli, že určité segmenty budou propojeny a vytvoří tak cyklus, nicméně nevíme jakým způsobem. Jediné, co by bylo možné usoudit, je to, že pokud počet navštívených segmentů byl menší než aktuální počet všech segmentů, potom by bylo puzzle chybné (vynucený krátký cyklus).

Dedukce na základě ep_1 a ep_2

Nyní máme určené body ep_1 a ep_2 a zjistíme si jednotlivá barevná pokračování v povolených směrech.

- *Dedukce 1*: Pokud nějaké barevné pokračování ep_1 vyúsťuje do ep_2 a počet navštívených segmentů je menší než aktuální počet všech segmentů, potom tato barva musí být nastavena na hodnotu křížku, jinak bychom dostali krátký cyklus.

- *Dedukce 2*: Nechť nějaké barevné pokračování ep_1 vyúsťuje do mřížového bodu x a nechť nějaké barevné pokračování ep_2 vyúsťuje rovněž do bodu x. Pokud dále je počet navštívených segmentů menší než aktuální počet všech segmentů, potom tyto dvě barvy nesmějí být zároveň obě čáry, jinak bychom dostali krátký cyklus. Jinými slovy tyto dvě barvy nesmějí mít zámek L. Stačí se podívat na mřížový bod x (ten tyto barvy sdílejí) a zjistit zámek mezi příslušnými hranami. Pokud povoluje možnost L, tak ji zakážeme.

ep1
ep2
ep1
x
ep2

Obrázek 2.15.3 - dedukce na základě koncových bodů

Zkoušení možných pokračování koncových bodů

V principu projdeme všechna možná pokračování koncového bodu segmentu (musí jít o povolený směr), podíváme se na barvu hrany v tomto směru, tu nastavíme v zásobníku barev na hodnotu čára a spustíme dedukce. Proces dedukcí je ukončen jedním z následujících stavů.

a) není možné nic dále vydedukovat – končíme s tím, že jsme na nic nového nepřišli
b) maximální počet barev v zásobníku byl překročen – sice lze dále dedukovat, ale zřejmě by tato možnost nevedla k „očividnému krátkému cyklu“, tak opět končíme
c) vydedukován spor – ten může nastat v tom případě, že bychom dosáhli krátkého cyklu nebo obecněji, že se snažíme nastavit barvě v zásobníku barev hodnotu opačnou než tu, kterou má již přiřazenu. Opět končíme, ale se zjištěním, že první zkoušená barva (neboli pokračování koncového bodu) musí mít hodnotu křížek.

Proces dedukcí probíhá následovně. Dokud není proces přerušen některým z výše zmíněných stavů, potom zkontroluj, zda by koncové body nemohly vytvořit cyklus a pokus se prodloužit koncové body. Vždy musíme oba koncové body kontrolovat v páru, jelikož prodloužení jednoho koncového bodu může mít za následek vynucení prodloužení druhého koncového bodu a naopak. Tato vynucení mohou probíhat „na střídačku“.

Kontrola oproti krátkému cyklu

Jde o jednoduchou kontrolu. Pouze se podíváme, zda jednotlivé koncové body nejsou od sebe vzdáleny o jedinou hranu. V kladném případě zjistíme, zda je počet navštívených segmentů (který při procesu dedukcí průběžně aktualizujeme) menší než aktuální počet všech segmentů. Pokud tomu tak je, potom barvu takovéto hrany nastavíme na hodnotu křížek (jinak bychom obdrželi krátký cyklus) a vložíme tuto informaci do zásobníku dedukcí.

Hledání prodloužení koncového bodu

Nejprve si klasifikujeme všechny hrany, které vstupují do koncového bodu do pěti kategorií.

- zakázané hrany – takové hrany, jež neleží v povoleném směru
- čáry
- křížky
- čáry na základě obarvení – jde o neurčené hrany, avšak jejich hodnotu lze na základě jejich obarvení přečíst v zásobníku barev hran
- křížky na základě obarvení – podobně jako výše

Všimněme si, že existuje ještě jedna kategorie a to neurčená hrana, která ani nemá určenou barvu v zásobníku barev. Pro účely algoritmu však nemá smysl tuto hodnotu uchovávat a případný počet takovýchto hran lze dopočítat z počtu hran v ostatních kategoriích.

Nyní již můžeme provést dvě triviální dedukce pro zajištění konzistence mřížových bodů.

1) pokud součet počtu čar a čar na základě obarvení je větší než jedna, pak jsme došli ke sporu. Hrana, po které jsme do tohoto mřížového bodu přišli, nebyla mezi povolenými hranami, a tedy byla klasifikována jako zakázaná. Z toho ale vyplývá, že počet čar v mřížovém bodě by byl větší než 2, což nesmí nastat.

2) pokud naopak součet počtu křížků a křížků na základě obarvení je roven celkovému počtu hran v mřížovém bodě (4) mínus počtu zakázaných hran, pak z toho ihned plyne, že by segment neměl kam pokračovat (byl by v tomto koncovém bodě uvězněn). Tedy taktéž jsme došli ke sporu.

Pokud práce algoritmu hledání prodloužení není ukončena nějakou z výše uvedených triviálních dedukcí, potom je tento mřížový bod v konzistentním stavu a můžeme hledat pokračování. Budeme rozlišovat následující situace.

***Situace 1** – počet čar je roven jedné*

Dostali jsme se do koncového bodu nějakého dalšího souvislého segmentu čar. Musíme nastavit všechny neurčené barvy hran v tomto mřížovém bodě na hodnotu křížek v zásobníku barev. Dále se musíme přesunout do druhého koncového bodu nového segmentu a určit povolené směry (obdobně jako bylo popsáno v preprocessingové části).

***Situace 2** - počet čar na základě obarvení je roven jedné*

Máme jednoznačně určeno pokračování. Nastavíme ostatní (neurčené) barvy hran v mřížovém bodě na hodnotu křížku a přesuneme se po hraně s obarvením s hodnotou čára do nového koncového bodu. Opět ještě nastavíme povolené směry.

***Situace 3** – součet počtu zakázaných hran, křížků a křížků na základě obarvení je roven třem*

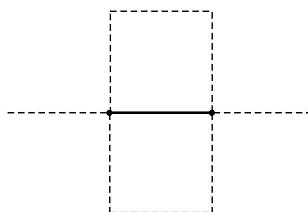
Pokračování je vynuceno okolními křížky – existuje právě jedna možnost, kudy lze opustit mřížový bod. Barvu takové hrany, po které lze mřížový bod opustit, nastavíme na hodnotu čára a přidáme do zásobníku barev. Přesuneme se do nového koncového bodu a nastavíme povolené směry.

***Situace 4** – umělé prodloužení*

Zjistíme všechna ta barevná pokračování v povolených směrech s neurčenou hodnotou hrany a její barvy (pokud byla hrana určena, tak musí jít o křížek, jinak bychom aplikovali Situaci 1, dále pokud je barva určená, pak jde taktéž o křížek – jinak aplikace Situace 2). Pokud zjistíme, že všechna vyúsťují do jediného mřížového bodu, potom jsme dostali umělé prodloužení. Přesuneme se sem a určíme povolené směry.

Pokud nenastane ani jedna z popsaných situací, tak segment nelze prodloužit a končíme.

V průběhu algoritmu nastavujeme barvám hran hodnoty (čára nebo křížek). Abychom toho mohli vydedukovat co nejvíce, používáme následující algoritmus.



Obrázek 2.15.4 – bezprostřední sousedé hrany

Nejprve hodnotu barvy uložíme do zásobníku barev. Pokud existuje barva opačná k této, potom ji rovněž přidáme do zásobníku barev avšak s opačnou hodnotou. Dále k hraně učíme jejích 8 základních sousedů (viz obrázek). Ke každému ze sousedů lze jednoduše zjistit zámek s nastavovanou hranou (ze stavu okolních mřížových bodů a okolních políček). Na základě hodnoty nastavované barvy hrany a zámku se sousedem, můžeme vydedukovat hodnotu barvy souseda následujícím způsobem.

Pokud hrana je čára a soused má:

- zámek A - barva souseda má hodnotu křížek
- zámek AC - barva souseda má hodnotu křížek
- zámek CL - barva souseda má hodnotu čára

Pokud hrana je křížek a soused má:

- zámek A - barva souseda má hodnotu čára
- zámek AL - barva souseda má hodnotu čára
- zámek CL - barva souseda má hodnotu křížek

Později bylo přidáno drobné vylepšení algoritmu. Při dedukcích pokračování z jednoho mřížového bodu děláme zároveň průnik shodných hodnot vydedukovaných barev. Pokud po projití všech pokračování tohoto mřížového bodu existuje nějaká barva, která měla ve všech případech nastavenou tutéž hodnotu, pak tuto hodnotu mít musí – a nastavíme ji.

Časová a paměťová složitost algoritmu

Pro realizaci algoritmu pokročilých dedukcí cest používáme frontu, do které ukládáme čísla reprezentantů pro kontrolu. Fronta je lineární vzhledem k počtu mřížových bodů a ostatním funkcím, zajišťujících chod algoritmu, stačí konstantní paměť.

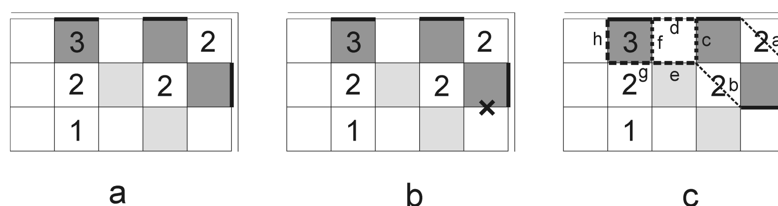
Poznamenejme, že tento algoritmus se provádí pouze v případě, že ostatními (jednoduššími) dedukcemi nelze nic zjistit. Dále se testuje pouze segment, který ještě nebyl kontrolován, což je zajištěno právě frontou. Do fronty se segment vrací pouze v případě sloučení dvou segmentů, potom jej lze znovu testovat. Díky frontě

tedy není zapotřebí prohledávání celého prostoru, ale rovnou máme určitý segment ke kontrole k dispozici.

Jedno spuštění dedukcí cest na určitém segmentu má konstantní amortizovanou složitost (ačkoliv tato konstanta není nejmenší – ale nezávisí na velikosti puzzle), díky tomu, že zkoušíme testovat maximálně 6 barev a každá z nich může vynutit vždy maximálně 19 dalších barev (díky omezení zásobníku barev – snaha lokalizovat problém krátkých cyklů). Celkový maximální počet spuštění dedukcí cest je n . log n, kde n je počet mřížových bodů.

Praktická ukázka

Podívejme se na následující obrázek (2.15.5). V části a je znázorněn výřez puzzle (pravý horní roh). Přestože to není na první pohled zřejmé (záleží na schopnostech hráče), lze vydedukovat stav jedné hrany (obrázek část b).



Obrázek 2.15.5 – ukázka detekce krátkého cyklu

Na obrázku v části c je vyobrazena situace, která by nastala, pokud by daná hrana byla čarou. Hrany označené písmeny a a b jsou uměle vzniklé hrany (musejí vést do daných mřížových bodů, jen zatím není jasné kudy přesně). Hrana c musí být křížek, jinak by nám vznikl krátký cyklus. Z toho implikuje, že hrany d a e musejí být čarami. Hrana f musí být křížek ze stejného důvodu, jako hrana c. Přesto obdržíme krátkou smyčku (díky vlastnostem nápovědy 3), jelikož hrany g a h musejí být čarami. Dojdeme tedy ke sporu.

2.16 VYLEPŠENÁ PROPAGACE PRAVIDEL V POLÍČKU

Motivace

Doposud spolu různá pravidla (např. normální a 3-zámky) komunikovala pouze prostřednictvím nastavení přípustných hodnot pro mřížové body. Pro nižší úrovně obtížnosti toto bylo postačující. Lze však dosáhnout lepší vzájemné komunikace.

Aplikací nějakého setu pravidel na políčko se u jednotlivých mřížových bodů vyřadí nepřípustné stavy (tj. ty, kterých nelze dosáhnout při zachování konzistentního stavu políčka). Pokud vezmeme pro každý ze čtyř mřížových bodů obklopujících políčko jeden přípustný stav, neznamená to ale, že máme přípustnou

kombinaci pro políčko. Jinými slovy může se stát (a častokrát tomu tak je), že tato čtveřice nesmí být spolu v kombinaci.

Idea algoritmu

Pro každé políčko si budeme navíc pamatovat množinu přípustných pravidel. Potom aplikace setu pravidel na políčko vypadá následovně. Nejprve zjistíme klasickým způsobem (dle přípustných stavů mřížových bodů a množiny zámků daného setu pravidel), zda lze pravidlo na políčko použít. Pokud ano, tak se ještě podíváme, zda lze toto pravidlo skutečně použít (tj. celá kombinace čtyř mřížových bodů je přípustná). V tom případě pravidlo skutečně aplikujeme a pokračuje obvyklým způsobem. Jestliže zjistíme, že pravidlo nelze aplikovat (klasickým způsobem), tak provedeme ještě test, zda celá kombinace mřížových bodů není nastavena jako přípustná. Pokud by byla, je zapotřebí nastavit nepřípustnost této kombinace a navíc nastavit příznak, že se množina aplikovatelných pravidel změnila. Jestliže po ukončení celé fáze aplikací pravidel v políčku je tento příznak nastaven, pak přidáme políčko zpět do fronty políček. Je zde možnost, že nějaký jiný set pravidel může z této změny vytěžit další dedukce.

Podmínkou pro využití této metody je jistý řád v pravidlech. Všechny sety pravidel byly vytvořeny se stejnou bázovou strukturou. Tou je čtveřice stavů mřížových bodů, zbytek bitů v pravidle záleží na konkrétním setu (jaké zámky či vztahy reprezentujeme). Tudíž například i-té pravidlo normálních dedukcí v políčku reprezentuje stejnou kombinaci čtyř mřížových bodů jako i-té pravidlo 3-zámků, kde $i \in \{1, \ldots, 160\}$. Jediný rozdíl spočívá v zachycení rozdílných skutečností téhož stavu.

Realizace

Vylepšenou propagaci realizujeme pomocí pole. Abychom šetřili pamětí, budeme s ním pracovat jako s bitovým polem. Pokud chceme zjistit, zda je j-té pravidlo aplikovatelné na políčko i, potom se podíváme na (160 * i + j)-tý bit v poli. Přepočet na skutečný index do pole, a jaký bit máme na této pozici vzít, realizujeme pomocí absolutního a relativního posunu.

3 ÚROVNĚ OBTÍŽNOSTI

3.1 JEDNOTLIVÉ ÚROVNĚ

Popsané metody vytvářejí relativně složitá puzzle, a to už od metod nejzákladnějších. Proto byla snaha naimplementovat taktéž jednodušší varianty hry. Program nakonec obsahuje 7 úrovní obtížnosti, přičemž u každé si může ještě zvolit, zda chce, aby generátor využíval techniku obarvování políček.

Velmi jednoduchá úroveň

Měla by sloužit pro úplné začátečníky, kteří se s touto hrou ještě nesetkali. Jediné dedukční techniky, které postačují hráči k vyřešení puzzle, jsou následující.

- dodržování nápověd – pouze dedukce typu: počet čar u políčka je roven nápovědě, ostatní hrany musejí být tedy křížky nebo naopak, kdy v políčku máme maximální počet křížků, což implikuje v doplnění čar
- dodržování mřížových bodů – pouze to, že zde musí být právě 0 nebo 2 čar. Dedukce jsou tedy například: mám 2 čáry, z toho plyne, že zbývající dvě hrany jsou křížky. Jiným příkladem může být situace, kdy v mřížovém bodě máme 1 čáru a 2 křížky. Zbylá hrana musí být čarou, jinak bychom zde uvěznili souvislý segment čar
- předejití krátké smyčce – pokud chybí jediná hrana, která by nám mohla uzavřít segment čar, a vznikl by tak krátký cyklus, pak tato hrana musí být křížek.

Pro tuto úroveň jsou implementována vlastní pravidla v políčkách a mřížových bodech. Základní verze předcházení krátké smyčky se neliší od jiných úrovní. Důležité však je, že u políček uvažujeme pouze 4 okolní hrany (nikoliv 12, jak je tomu u vyšších úrovní). Počet pravidel je potom nižší – stačí nám 15 pravidel – a tedy i generování puzzle je daleko rychlejší. Navíc dedukce se provádějí pouze v políčkách s nápovědou. Každé takové puzzle obsahuje alespoň jednu nápovědu 0, aby bylo možné začít dělat výše naznačené dedukce. Je zde možnost také využívat techniky obarvování políček, avšak pouze triviální dedukce. Ze znalosti vztahu barev dvou sousedních dedukujeme stav hrany (čára/křížek) a naopak ze stavu hrany dedukujeme vztah mezi barvami.

Jednoduchá úroveň

Tato úroveň používá techniky stejné jako velmi jednoduchá úroveň, navíc rozumí základní verzi zámků. Zámky uvažujeme pouze dvoustavové – Locked/Antilocked, přitom uvažujeme jen zámky mezi dvojicí hran, které jednak sdílí společný mřížový bod, a také jsou navzájem kolmé (svírají úhel 90°). To přesně odpovídá způsobu myšlení hráče, který se se zámky seznamuje a nemá rozvinuty větší praktické dovednosti pro řešení puzzle. Opět platí, že u políčka uvažujeme jen 4 hrany a nededukujeme u políček bez nápovědy. Pokud využijeme obarvování, potom lze dedukovat navíc i ze zámků. V políčkách jsou uváženy dva

zámky mezi příčnými hranami – ty slouží pouze pro dedukce nad obarvením políček a nemohou nijak propagovat dále. Pravidla mřížových bodů i políček mají poněkud odlišnou strukturu. První část obsahuje 4 dvojice bitů, které udávají, zda na daném místě může být hrana a křížek. Pokud nějakou hranu máme určenou, tak se provedou restrikce. Druhá část obsahuje opět 4 dvojice bitů, udávajících možné stavy zámků, mezi sousedícími hranami.

Pokročilá úroveň

Nakonec byla přidána ještě jedna úroveň, tak aby zohledňovala schopnosti hráčů. Rozdíl mezi jednoduchou úrovní a normální byl totiž propastný. Používáme stále jen 4 hrany okolo políčka, a dedukujeme jen v políčkách s nápovědami. Zavádíme však CAL zámky, které mají obrovský potenciál při řešení úloh (viz demonstrace CAL zámků v Dodatku 10.2)

Normální úroveň

Využívá CAL zámky a normální dedukce v políčku. Pracuje s 12 hranami u políčka, což umožňujících větší propagaci. Navíc dedukce provádíme i nad políčky bez nápovědy.

Obtížná úroveň

Implementuje 3-zámky a diagonální zámky.

Velmi obtížná úroveň

Využívá navíc techniky obarvování hran a kontrolu oproti krátkým barevným cyklům. Obarvování hran dedukuje taktéž z diagonálních zámků.

Úroveň expert

Využívá pokročilé dedukce cest. Navíc využívá pokročilá pravidla nad obarvováním políček i hran. Mezi pravidly používá mechanismus vylepšené propagace. Puzzle vygenerované touto úrovní je již velmi obtížné i pro zkušeného hráče. K jeho vyřešení je zapotřebí několikeré použití fixování pozic. Obtížností se dokáže vyrovnat i puzzle vygenerovaným pomocí backtrackingu, avšak rozdíl v čase pro vygenerování je závratný (díky polynomiální časové složitosti).

Důležitá poznámka

Pro každou úroveň platí, že je schopna vyřešit veškeré puzzle vygenerované jakoukoliv nižší úrovní. Kvůli efektivitě algoritmu není však zaručen opačný vztah. Může se stát totiž, že vygenerované puzzle lze vyřešit pomocí jednodušší úrovně. Takováto situace se však stává zřídkakdy. Pravděpodobnost, že se v puzzle alespoň jednou použije metoda dané úrovně a to tak, že nižší úroveň ji nedokáže nahradit složením jednodušších dedukčních metod, je velmi vysoká. Proto po vygenerování neprovádím kontrolu na skutečnou obtížnost a v případě neúspěchu negeneruji nové puzzle.

3.2 OBECNĚ K OBTÍŽNOSTI PUZZLE

Obtížnost puzzle je čistě subjektivní záležitost. Záleží na schopnostech a zkušenostech konkrétního hráče. Většině hráčů bohatě stačí pokročilá úroveň. Úroveň expert by pak měla postačovat i hráčům velmi zkušeným. Samozřejmě, že existují i mnohem těžší puzzle, avšak pro většinu lidí již nejsou nijak zábavná. Jsou vytvořena pomocí plně rekurzivního backtrackingu a jejich vyřešení spočívá ve zkoušení jednotlivých hran. Po dlouhé době hraní můžeme dojít ke sporu. Zjistíme tím, že hrana musí mít opačnou hodnotu a můžeme začít zkoušet další. A navíc čas pro vygenerování takového puzzle je neúnosně dlouhý.

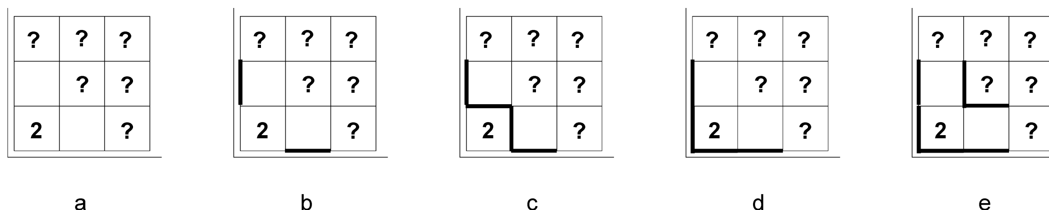
Obtížnost puzzle také záleží na zastoupení nápověd. Například pokud se v puzzle vyskytují pouze nápovědy 1 a 2, pak je obvykle mnohem obtížnější než puzzle stejné úrovně při zastoupení všech druhů nápověd.

Setkal jsem se také s názorem (od zkušených hráčů na fóru kwontomloop.com [7]), že puzzle vygenerovaná mým programem jsou zábavnější než puzzle generována backtrackingem. Je to nejspíš způsobeno tím, že program v podstatě emuluje způsob řešení člověka. Dedukce dělá pouze tam, kde se stala nějaká změna.

3.3 HIGHLANDER DEDUKCE

Jedná se o metodu při řešení puzzle těžící z jednoznačnosti smyčky. Metoda pravděpodobně vznikla na fóru kwontomloop.com [7] a její název pochází z výroku *„There can only be one!“*. Pro vysvětlení použiji stejný příklad, na jakém to bylo vysvětleno mě uživatelem MondSemmel na již zmíněném fóru.

Podívejme se na následující obrázek 3.3. V části a je zachycena část puzzle, konkrétně levý dolní roh. Podstatné však je, že prázdná políčka musejí být opravdu prázdná, políčka s otazníkem mohou obsahovat cokoliv (prázdná či libovolná nápověda).



Obrázek 3.3 – highlander dedukce

Pomocí zámků jednoduše vydedukujeme dvě hrany, tak jak je naznačeno v části b. Dále existují dvě možnosti, jak vinout smyčku okolo nápovědy s číslem dva (viz část c a d). Předpokládejme, že situace v části c je korektní. Potom ale lze

na místo toho vždy použít situaci znázorněnou na obrázku 3.3 v části d. Pokud by to tedy tak bylo, puzzle by mělo dvě řešení. Z toho implikují dvě hrany v pravém horním rohu u nápovědy s číslem 2 (viz část e).

Tuto metodu lze využít pouze u puzzle, kde máme zaručenou unikátnost řešení. Vycházíme z toho, že kdykoliv nalezneme situaci, kde existuje více jak jedna možnost, jak tuto situaci vyřešit, potom všechny tyto možnosti jsou špatně.

Znamená to tedy, že pouhá snaha snižovat počet nápověd při generaci puzzle, nemusí nutně znamenat těžší puzzle. Občas tomu bývá i naopak – nedostatek nápověd implikuje využívání tohoto typu dedukcí a puzzle se stává snadnějším.

Tuto dedukční metodu nepoužívám ve svém algoritmu pro řešení. V čase generování puzzle totiž nelze ještě předpokládat jednoznačnost.

4 GENERÁTOR PUZZLE

4.1 GENEROVÁNÍ SMYČKY

Motivace

Algoritmus pro generování smyčky, který jsem navrhl a implementoval, je celý založen na topologii. Hlavní přínos spočívá v tom, že odpadá složitá práce s hranami, a soustředíme se pouze na políčka, která mohou nabývat dvou stavů – mohou ležet uvnitř nebo vně smyčky.

Pro generaci dobrého puzzle je zapotřebí vytvořit zajímavou smyčku. Žádné pravidlo, jak by taková smyčka měla vypadat, bohužel neexistuje. Proto jsem položil tři základní požadavky na smyčku:

- měla by vyplňovat téměř celý prostor
- měla by být dostatečně klikatá
- neměla by být předvídatelná

První požadavek je samozřejmý, jelikož by bylo nežádoucí mít smyčku jen na polovině hrací plochy nebo dokonce na menší části. Avšak postačující není. Představme si smyčku, která leží na obvodu obdélníkové hrací plochy, potom všechna políčka leží uvnitř smyčky. Ačkoliv je první požadavek splněn, přesto se očividně nejedná o zajímavou smyčku.

Idea algoritmu

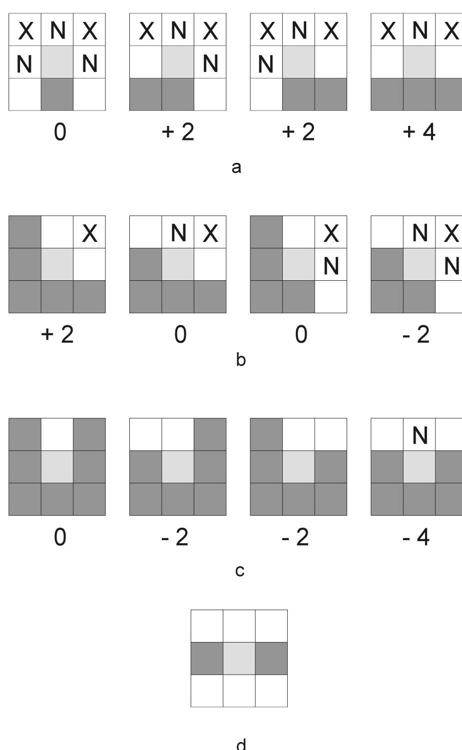
O políčku řekneme, že je ve stavu OFF, pokud leží vně smyčky, a ve stavu ON, pokud leží uvnitř. Před začátkem chodu algoritmu jsou všechna políčka ve stavu OFF. Ideu lze zjednodušeně popsat tak, že náhodně vybereme políčko na hrací ploše, nastavíme mu příznak ON a vznikne tak nejmenší možná smyčka. Dále pokračujeme inkrementálním způsobem. V každém kroku algoritmu přidáme jedno políčko a rozšíříme tím smyčku. Musíme však dodržovat určitá pravidla, abychom docílili předem nastíněných požadavků.

Nechť jsou rozměry hrací plochy m x n. Otázkou je, jak zaručit, aby smyčka vyplňovala téměř celý prostor a zároveň byla dostatečně klikatá a nepředvídatelná. Nasnadě je použít jako kritérium počet políček, které budou uvnitř smyčky (budou ve stavu ON). Ovšem toto jednoduché řešení vůbec nemusí splňovat naše požadavky. Jednak vůbec není zaručena klikatost a dále i přesto, že například zvolíme, že minimální počet políček ve stavu ON by měl být (m x n) / 2, tak může být vyplněna pouze jedna polovina hrací plochy.

Moje řešení spočívá ve splnění minimálního počtu zatáček pro danou velikost hrací plochy. Zatáčku definujeme jakožto mřížový bod, ve kterém se mění směr.

Při práci algoritmu budeme uvažovat hranici okolo smyčky. Hranicí budou veškerá políčka ve stavu OFF, která jsou bezprostředními sousedy některého z políček uvnitř smyčky (stav ON). Smyčka se může rozrůst vždy pouze o políčko z hranice. Ne každé takovéto políčko je ale vhodný kandidát pro rozšíření. Musíme si dávat pozor na dvě věci – zaprvé, aby se nám smyčka nikde nekřížila, a zadruhé, abychom měli stále jen jedinou smyčku (aby nevznikly další).

V průběhu algoritmu platí následující invariant. Smyčka po rozšíření o jedno políčko je vždy v korektním stavu (nekříží se, existuje jediná smyčka). Pravidla pro rozšiřování smyčky záleží na počtu sousedů ve stavu ON. Na obrázku 4.1.1 si přiblížíme veškerá potřebná pravidla.



Obrázek 4.1.1 – pravidla pro rozšiřování smyčky

Tmavá barva na obrázku značí políčka ve stavu ON (uvnitř smyčky), a bílá označuje políčka ve stavu OFF. Světlá barva udává políčko na hranici, které dosud bylo ve stavu OFF a my se pokoušíme o jeho přidání ke smyčce. Zaměřme se nejprve na obrázku 4.1.1 na část a. Zde jsou vyobrazena pravidla pro přidávání políčka, pokud sousedí pouze s jedním políčkem ve stavu ON. Nutná podmínka pro přidání je, aby políčka označena symbolem X byla ve stavu OFF. Pokud by totiž byla ve stavu ON, vzniklo by nám křížení smyčky. Symbol N nám potom značí nová políčka hranice, tedy ta, která ještě na hranici nebyla. Zobrazena je pro jednoduchost pouze jedna ze čtyř rotačních symetrií. Pod každým z pravidel je ještě uveden číselný údaj. Ten značí, o kolik se změní počet zatáček přidáním nového políčka ke smyčce. Na obrázku 4.1.1 v části b jsou zobrazena pravidla pro políčka se dvěma sousedy ve stavu ON, a v části c se třemi sousedy ve stavu ON.

Povšimněme si, že nově přidávané políčko nemůže mít 0 sousedů (jinak by neleželo na hranici), a ani 4 sousedy (měli bychom alespoň dvě smyčky).

Nakonec v části d na obrázku 4.1.1 je znázorněna jediná situace, které se musíme vyvarovat. Jde o políčko, jež má 2 sousedy ve stavu ON a zároveň tyto sousedé mají rozmístění jako na obrázku 4.1.1. Označme políčko vlevo od středového jako A, a políčko vpravo jako B. Obě tato políčka jsou uvnitř smyčky (jelikož jsou ve stavu ON). Z Jordanovy věty o kružnici plyne, že existuje oblouk, který spojuje A a B a celý leží uvnitř smyčky. V této situaci si to lze představit tak, že existuje z A do B souvislá cesta po políčkách ve stavu ON. Přidáním světlého políčka vznikne druhá cesta z A do B, a tedy dostaneme (topologickou) kružnici. Jinými slovy bychom dosáhli nechtěně další smyčky.

Realizace

Budeme potřebovat pole o velikosti m x n pro uchování informací zda jsou políčka ve stavu ON či OFF. Dále budeme potřebovat nějakou datovou strukturu pro záznam políček na hranici. Jako optimální struktura se ukázala halda. Uzly v haldě budou mít následující tvar – jednak budou obsahovat data (pozice políčka na hranici), dále pak klíč. Klíčem bude náhodně vygenerované číslo, což nám umožní, aby kandidáti pro přidání ke smyčce byli vybíráni náhodně po celém obvodu hranice (jinak by se mohla projevit tendence rozšiřovat smyčku jen určitým směrem).

Na základě empirických pokusů jsem odvodil minimální počet zatáček potřebný pro vygenerování zajímavé smyčky. Je dán vztahem C x m x n, kde C = 0,65. Pokud by bylo toto číslo menší, tak nedocílíme vyplnění celé hrací plochy a smyčka nebude ani dostatečně klikatá. Větší hodnota této konstanty vede ke dvěma nepříznivým důsledkům. Jedním je fakt, že algoritmus častokrát neuspěje, jelikož v určitém stádiu, při přidávání políček, na hranici již není žádné takové, které by mohlo počet zatáček zvýšit. Halda se vyprázdní a algoritmus končí neúspěchem. Druhý důsledek je jistá předvídatelnost tvaru smyčky při řešení puzzle. Pokud bychom se snažili o maximální klikatost smyčky, zjistíme, že smyčka sice bude klikatá, ale zároveň úzká (ve všech místech bude mít šířku jedno, maximálně dvě políčka).

Pro porovnání program LoopDeLoop [5] nesplňuje požadavek pro vyplnění prostoru (častokrát se zde objevují obrovská místa hrací plochy vyplněná nulami), program Loopy [6] naopak nesplňuje požadavek nepředvídatelnosti (úzká klikatá smyčka).

Popis algoritmu

Před vlastním popisem si ještě charakterizujeme kandidátská políčka pro přidání (na hranici) podle počtu sousedů ve stavu ON. Kandidáti s jedním sousedem mají tendenci smyčku prodlužovat do délky. Splňují tedy první požadavek, tj. aby se nám smyčka rozprostřela co nejvíc po hrací ploše. Zároveň mají neklesající charakter co do počtu zatáček (po řadě 0, +2, +2, +4). Avšak pokud bychom používali jen tato pravidla, dosáhli bychom úzké smyčky, což přináší nepříznivý efekt předvídatelnosti. Kandidáti se dvěma sousedy ve stavu ON mají v podstatě

neutrální dopad na počet zatáček (-2, 0, 0, +2), zvyšují však robustnost smyčky. Nakonec kandidáti se třemi sousedy ve stavu ON mají tendenci snižovat počet zatáček a zvyšovat robustnost.

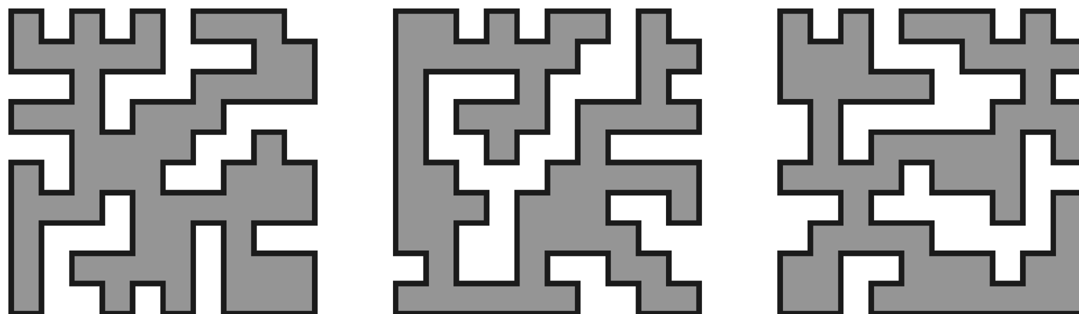
Definujme p_1, p_2 a p_3 jako pravděpodobnosti, že políčka na hranici, která mají po řadě jednoho, dva, tři sousedy ve stavu ON, budou ke smyčce opravdu přidána poté, co splňují nutné podmínky pro přidání. Pomocí těchto parametrů lze dosáhnout různých tvarů smyček (úzkých, robustních apod.)

Algoritmus začíná iniciační částí a poté následují tři fáze. První fáze je zásadní. Rozhoduje nám o úspěchu či neúspěchu vygenerovat smyčku. K chodu budeme potřebovat dvě haldy. V první fázi začneme tím, že náhodně vybereme políčko někde ze středu hrací plochy - bude tvořit jakýsi zárodek smyčky. Pokud bychom vybrali políčko někde u okraje hrací plochy, potom by hrozilo velké riziko, že se nám smyčka nerozprostře po celé hrací ploše (nedostane se až ke druhému okraji). Do první haldy vložíme čtyři sousední políčka. Ta nám tvoří hranici. Dále postupujeme tak, že dokud není halda prázdná, vyzvedneme z ní jedno hraniční políčko. Pokud má více jak jednoho souseda ve stavu ON, prozatím se jím nezabýváme a vložíme jej do druhé haldy. Pokud má právě jednoho souseda ON, zavoláme metodu, která zjistí, zda lze políčko skutečně přidat. Zároveň nám oznámí nová hraniční políčka pro přidání do haldy a taktéž změnu údaje o celkovém počtu zatáček. V případě úspěchu si ještě vygenerujeme náhodně číslo určující pravděpodobnost, a pokud je menší než p_1, tak políčko skutečně přidáme. Musíme upravit hrací plochu (nastavit políčku stav ON), dále změnit počet zatáček a nová hraniční políčka vložit do haldy. Po vyčerpání haldy zkontrolujeme, zda je celkový počet zatáček větší než minimálně určená hranice. Pokud ne, tak se první fáze opakuje do té doby, než uspěje. V případě úspěchu pokračujeme druhou a třetí fází. Nejprve však obsah druhé haldy přesuneme do první haldy, a druhá bude sloužit opět jako záložní. Tyto fáze jsou obdobné – přidáváme políčka ke smyčce se dvěma a třemi sousedy ve stavu ON v závislosti na pravděpodobnostech p_2, p_3. Fáze 2 a 3 již na úspěšnost generace nemají vliv (může se stát, že se nám počet zatáček sníží až pod minimální hranici, avšak vhodnou volbou p_2 a p_3 lze tomu zabránit).

Algoritmus má pravděpodobnostní charakter, tudíž nelze zaručit dobu běhu. Z empirických pokusů však vyplývá, že při vhodně nastavené konstantě (C = 0,65) a parametrech p_1, p_2, p_3 nám i pro velká puzzle (40 x 30) stačí maximálně dva, či tři pokusy v první fázi. Pro zaručení úspěchu si algoritmus převzorkovává parametr p_1 tak, aby jeho minimální hodnota byla 0,8 (tedy původní pravděpodobnost se z intervalu [0; 1] zobrazí na interval [0,8; 1]).

Praktická ukázka

Na obrázku 4.1.2 jsou vyobrazeny tři náhodně vygenerované smyčky na hrací ploše o rozměrech 10x10 s parametry ($p_1 = 1$; $p_2 = 0{,}3$; $p_3 = 0{,}3$).



Obrázek 4.1.2 – ukázka vygenerovaných smyček

4.2 GENEROVÁNÍ PUZZLE

Idea algoritmu

Generátor dostane na vstupu výšku a šířku puzzle, které má vytvořit. Dalšími parametry jsou obtížnost puzzle, dále zda používat deduktivní metodu obarvování políček, jaké hodnoty nápověd používat a nakonec typ symetrie. Zjednodušeně lze algoritmus popsat takto – nejprve vygenerujeme smyčku pro hrací plochu dané velikosti. Poté vytvoříme puzzle dle této smyčky, kde všechna políčka budou nápovědou (budou obsahovat číselný údaj). V náhodném pořadí projdeme veškerá políčka, a pokud lze puzzle vyřešit i bez nápovědy v určitém políčku, tak nápovědu odstraníme.

Zamíchání pořadí nápověd pro odstranění

Vytvoříme pole, kde jednotlivé prvky jsou pozice políček v pořadí, ve kterém se budou procházet a pokoušet odstraňovat. Nechť A je toto pole a nechť N je počet políček. Na počátku platí $A[i] = i$, pro $\forall i \in \{1, \ldots, N\}$. Zamíchání nápovědami, neboli náhodné přeskupení prvků v poli A, realizujeme pomocí Durstenfeldovy verze algoritmu Fisher-Yates shuffle (viz článek [8]) v čase $O(|N|)$.

Povolené hodnoty nápověd

Taktéž lze nastavit povolené hodnoty nápověd. Například můžeme chtít vygenerovat puzzle, kde nápovědy mohou být pouze čísla 1 a 2 (tato kombinace většinou činí puzzle mnohem obtížnějším). Ovšem ne každá kombinace hodnot je vhodná k vytvoření puzzle. Pro určitou úroveň obtížnosti a množinu povolených hodnot nápověd se vůbec nemusí podařit puzzle vygenerovat. Algoritmus je modifikován oproti obecné verzi tak, že po vygenerování smyčky rovnou odstraníme všechny nepovolené nápovědy a pokusíme se puzzle vyřešit. Pokud to lze, tak se snažíme odstraňovat zbylé nápovědy stejným způsobem, jako u původní verze algoritmu. V opačném případě vygenerujeme novou smyčku a pokusíme se opět puzzle vyřešit po odstranění nápověd (tedy postupujeme tak, jak bylo uvedeno výše).

Symetrie

Implementuji dva typy symetrií – osovou a středovou. Pokud je zvolena osová symetrie, potom pro každé políčko, které obsahuje nápovědu, platí, že obraz políčka vzhledem k ose x obsahuje taktéž nápovědu a zároveň obraz políčka vzhledem k ose y obsahuje nápovědu. Vzniká tak obecně čtveřice políček, která jsou buď všechna s nápovědou, nebo všechna bez nápovědy. U středové platí to, že políčko má svůj obraz vždy v pootočení o 90°. Tedy jsou zde taktéž políčka tvořící semknuté čtveřice.

Při použití symetrií vznikají jednodušší puzzle (vzhledem k obtížnosti). Je to způsobeno tím, že musíme odstranit buď celou čtveřici políček, nebo žádné. I přestože při odstranění například třech z nich by bylo puzzle stále řešitelné.

Na obrázku 4.2.1 jsou ukázky symetrií – v části a osová symetrie a v části b středová symetrie. Poláčka s nápovědami jsou pro zvýraznění označena světle.

3	2	1		1	2		1	2	2
	3	1					0	0	
				2	0				
3		2	1	2	2	2	1		3
1									2
2									1
2		2	3	2	2	1	2		3
				2	3				
	2	3					0	0	
3	2	3		3	2		1	1	2

a

3	2			2		1	3	3	3
2	2			1	3	2	1	2	1
2	3			2					
1	2		1		2	3			
	3		1				1	2	3
1	2	2				3		3	
			2	2		2		1	3
					3			2	3
2	2	3	2	1	2			1	2
2	2	2	2		3			3	3

b

Obrázek 4.2.1 - příklady symetrií

Upozornění

Existují taková puzzle, která, i přestože všechna políčka obsahují nápovědu, nemají jednoznačné řešení. Takovýchto puzzle je malé množství, přesto si ale na ně musíme dát pozor. Příklad je uveden na následujícím obrázku (4.2.2). V algoritmu je to ošetřeno tak, že generování smyčky probíhá ve smyčce. Pokusíme se vyřešit puzzle se všemi nápovědami, a pokud uspějeme, pokračujeme obvyklým způsobem v odstraňování nápověd. V opačném případě smyčku zahodíme a vygenerujeme novou.

2	2	2	2
3	2	3	2
2	2	2	2
3	2	2	2

2	2	2	2
3	2	3	2
2	2	2	2
3	2	2	2

2	2	2	2
3	2	3	2
2	2	2	2
3	2	2	2

Obrázek 4.2.2 – nejednoznačnost puzzle

5 IMPLEMENTACE

5.1 POUŽITÁ TECHNOLOGIE

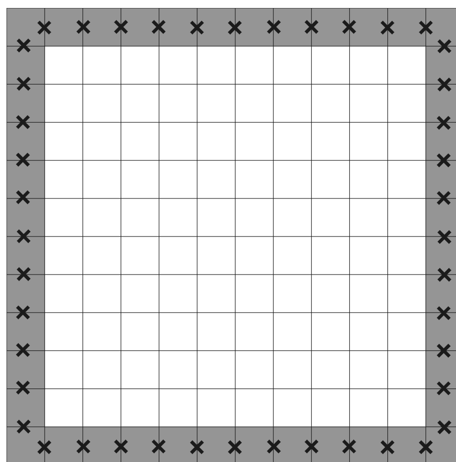
Program je implementován na platformě .NET, konkrétně v programovacím jazyce C#. Výhodou je snadné provázání kódu s grafickým rozhraním (GUI) pomocí WinForms. Nevýhoda pak spočívá v tom, že program běží pouze pod operačními systémy Windows od firmy Microsoft. Navíc pro svůj běh potřebuje Microsoft .NET Framework verze 3.5 SP1, který nemusí být na mnoha počítačích nainstalován a je zapotřebí jeho doinstalace. Další nevýhodou je fakt, že platforma .NET využívá tzv. *just-in-time* překládany kod, to znamená, že při kompilaci je kód přeložen do tzv. mezikódu a dodatečný překlad se provádí až při spuštění aplikace na uživatelově systému pomocí .NET Runtime. V praxi to má za důsledek pomalejší kód než je u nativně překládaných programů (např. C++).

Podporované operační systémy:

Microsoft Windows XP
Microsoft Windows 2000
Microsoft Windows Vista

5.2 REPREZENTACE HRACÍ PLOCHY

Je zapotřebí uchovávat informace o velikosti hrací plochy (výška, šířka), nápovědách, stavech políček, stavech mřížových bodů, a nakonec o stavech hran. Pro využití dalších metod, jako je například obarvování políček či hran, musíme uchovávat také tyto údaje.



Obrázek 5.2 – schéma hrací plochy o rozměrech 10x10 včetně zarážky

Pro snazší práci algoritmu si uchováváme hrací plochu o něco větší, vytvoříme tím jakousi zarážku okolo. Hrany reprezentujeme v jednorozměrném poli s tím, že nejprve tam ukládáme všechny horizontální hrany a poté vertikální. Tento způsob šetří paměť, potřebuje však při práci používat triviální přepočet (například pro zjištění okolních hran určitého políčka).

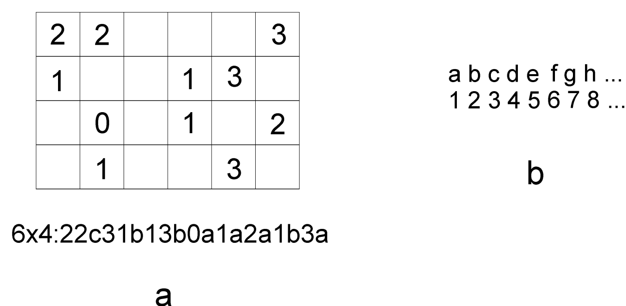
5.3 PRIORITNÍ FRONTA

Prioritní frontu implementuji na bázi min-haldy. Ukládají se do ní pozice (políček, mřížových bodů či jiných objektů). Na počátku je fronta inicializována maximálním počtem objektů (např. počtem políček). Fronta je navíc obohacena o (bitové) pole příznaků. Ty udávají, zda konkrétní objekt již ve frontě je či nikoliv. Díky častému lokálnímu charakteru při řešení puzzle (ovlivněné políčko ovlivní souseda, ten opět souseda, přičemž to může být opět původní políčko) je toto opatření vhodné. Zabraňuje vícenásobnému vložení některého objektu do fronty. Navíc pokud bychom opatření nezavedli, tak by velikost fronty mohla vzrůst daleko nad maximální počet objektů.

5.4 POPIS PUZZLE POMOCÍ ŘETĚZCE LOOPY

Puzzle lze popsat pomocí speciálního řetězce, který se poprvé objevil v aplikaci Loopy [6]. Později se začal používat i v jiných aplikací a na fórech k popisu puzzle a stal se z něj nepsaný standard. Má jednoduchý a úsporný tvar

Řetězec má formát WxH:string, kde W je šířka a H výška puzzle oddělené znakem x. Poté následuje dvojtečka a samotný řetězec. Ten popisuje políčka po jednotlivých řádcích směrem zleva doprava. Pokud políčko obsahuje nápovědu, tak jeho číselnou hodnotu zapíšeme do řetězce. V případě, že políčko nápovědu neobsahuje, pak záleží na počtu bezprostředně jdoucích takovýchto políček (bez nápovědy). Pokud je pouze jediné políčko bez nápovědy mezi dvěma políčky s nápovědou, pak do řetězce zapíšeme písmeno a. Jestliže by byla takováto políčka dvě, zapíšeme do řetězce b, a tak podobně. Jedná se tedy o ordinální číslo vzhledem k malým znakům anglické abecedy. Nejlépe situaci objasní obrázek 5.4.



Obrázek 5.4 – řetězec Loopy

Na obrázku 5.4 v části a je příklad puzzle a pod ním řetězec Loopy, jenž jej popisuje. V části b u obrázku 5.4 je pak naznačen převodní vztah mezi počtem políček bez nápovědy a znakem, který tento počet reprezentuje. Předpokládá se vždy maximálně jedno písmeno anglické abecedy mezi políčky s nápovědou (kvůli úspoře místa). Lze však (pro jednoduchý převod) pouze nahradit každé políčko bez nápovědy znakem a. Potom by Loopy řetězec z části a u obrázku měl tvar 6x4:22aaa31aa13aa0a1a2a1aa3a.

Musí však platit následující formule (N je délka řetězce).

$$\sum_{1}^{N} \delta(x_i) = W.H \text{ , kde } \delta(x_i) = \begin{cases} 1 & x_i \in \{0,\ldots,3\} \\ ord(x_i) & x_i \in \{a,\ldots,z\} \end{cases}$$

Důležitá poznámka

V době zpracování úlohy Slitherlink, aplikace Loopy podporovala pouze regulární hrací plochu, tedy čtvercová políčka s nápovědami 0-3. Nyní je tato aplikace dostupná v rozšířené verzi, kde je možnost výběru tvaru políčka – trojúhelníky, šestiúhelníky, atd. Z tohoto důvodu se také změnil Loopy formát. V aplikaci používám import / export v původním tvaru řetězce – stále se jedná o nejzažitější variantu.

5.5 SCHÉMA ALGORITMU PRO ŘEŠENÍ PUZZLE

Pro jednodušší úrovně puzzle, konkrétně velmi jednoduchou, jednoduchou a pokročilejší, je schéma obdobné tomu, které bylo uvedeno v kapitole 2.4. Pro vyšší úrovně je schéma následující. Uvažujeme úrovně normální (N), obtížnou (O), velmi obtížnou (VO) a expert (E) v tomto pořadí dle vzrůstající obtížnosti. U každé metody je uveden znak, který značí, že tato metoda se používá právě až od této úrovně, tj. nižší úrovně tyto metody nepoužívají. Zároveň je zde ještě možnost (B) označující použití technik založených na obarvování políček.

Všechny dedukce jsou prováděny pouze tehdy, pokud je úroveň nastavena na hodnotu uvedenou v závorce, nebo vyšší, v případě obarvování, pokud je zapnuto.

Znázornění algoritmu je k vidění na schématu 5.5.

```
dokud je fronta mříž. bodů a políček neprázdná
{
	dokud je fronta mříž. bodů a políček neprázdná
	{
		dokud je fronta mříž. bodů a políček neprázdná
		{
			dokud je fronta mřížových bodů neprázdná
			{
				vyzvedni mřížový bod
				proveď v něm normální dedukce (N)
				proveď dedukce nad obarvování políček (B)
				proveď dedukce nad obarvováním hran (VO)
			}
			jestliže je fronta políček neprázdná
			{
				vyzvedni políčko
				proveď v něm normální dedukce (N)
				proveď dedukce nad obarvování políček (B)
				proveď pokročilé dedukce nad obarvování políček (E)
				proveď dedukce nad 3-zámky (O)
				proveď dedukce nad diagonálními zámky (O)
				proveď dedukce nad obarvováním hran (VO)
				proveď diagonální dedukce nad obarvováním hran (VO)
				proveď pokročilé dedukce nad obarvováním hran (E)
			}
		}

		dokud je fronta barev neprázdná a fronta mříž. bodů a políček prázdná
		{
			vyzvedni barvu a otestuj ji na cyklus
		}
	}

	dokud je fronta reprezentantů neprázdná a fronta mříž. bodů a políček prázdná
	{
		vyzvedni reprezentanta a proveď dedukci cest
	}
}
```

Schéma 5.5 – algoritmus pro řešení puzzle

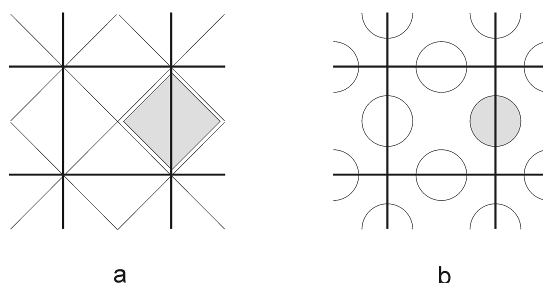
6 GRAFICKÉ UŽIVATELSKÉ ROZHRANÍ

Základní poznatky

Grafické uživatelské rozhraní (GUI) je podstatnou částí aplikace. Příjemný vzhled a pohodlné ovládání totiž z velké části ovlivňuje oblíbenost aplikace u hráčů.

Pokud jde o vykreslování hrací plochy, tak je realizováno pomocí double-bufferu. Z důvodu efektivity se vždy překresluje nejmenší potřebná část hrací plochy (například hrana při změně stavu a její malé okolí).

Uživatel si může zvolit způsob ovládání hry (klávesnice či myš). Zajímavé je ovládání myší a možnost jeho urychlení. Podívejme se na obrázek 6.1. V části a je zobrazen výřez hrací plochy a oblast (zvýrazněna šedou barvou), která odpovídá aktivní oblasti hrany. Toto schéma se využívá při kliknutí na hrací plochu pro určení ovlivněné hrany. Hru však lze urychlit tak, že povolíme tažení myši, což umožňuje nakreslení souvislého segmentu čar či křížků na jediné kliknutí. Pro správný chod musíme při tažení uvažovat menší akční radius hrany (viz obrázek 6.1b).



Obrázek 6.1 – schéma hrací plochy pro ovládání myší

Kontrola stavu puzzle na hrací ploše

V průběhu hraní hry je zapotřebí kontrolovat stav na hrací ploše (při každé události měnící tento stav) - potřebujeme oznámit krátký cyklus nebo fakt, že už jde o řešení. Algoritmus pro kontrolu puzzle je založen na topologii.

Idea algoritmu

Ke každému políčku určíme jeho komponentu souvislosti. Dvě sousední políčka jsou ve stejné komponentě, pokud hrana mezi nimi je neurčená nebo jde o křížek. Poté se rozhodujeme podle celkového počtu komponent.

1 komponenta – znamená, že ještě nemáme žádnou uzavřenou smyčku
2 komponenty – máme právě 1 smyčku, musíme testovat dál
3 a více komponent – alespoň dvě smyčky – jde o chybný stav

Pokud tedy máme dvě komponenty, je zapotřebí zjistit, zda se zde nenalézá nepřípustná hrana. To může být jak čára způsobující větvení, nebo čára úplně mimo smyčku. Test spočívá v tom, že pokud je hrana čarou, tak se podíváme na její dvě sousední políčka. Pokud jsou ve stejné komponentě, jde o chybu a nemůže jít o řešení.

V případě, že nenalezneme žádnou ilegální hranu, stačí již jen zkontrolovat nápovědy na přesné splnění. Mřížové body testovat nemusíme – jejich korektnost nepřímo plyne ze splnění předchozích částí.

Uvedený algoritmus předpokládá jednoznačnost puzzle. Nikterak nekontroluje, zda neexistuje i jiné řešení, jelikož by to bylo časově neúnosné v této situaci.

Další možnosti aplikace

Aplikace podporuje import/export puzzle pomocí řetězce Loopy, možnost obarvovat políčka, možnost zvětšovat/zmenšovat zobrazení puzzle, vícenásobné Fix/Undo, možnost ukládat a znovu načítat hru, tisk puzzle.

7 MOŽNÁ ROZŠÍŘENÍ

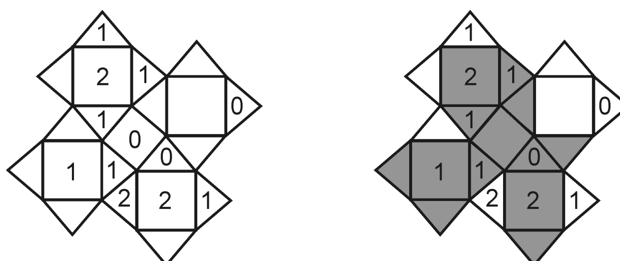
7.1 DEDUKCE NA ZÁKLADĚ CYKLŮ OBARVENÍ POLÍČEK

Jedná se o možné vylepšení algoritmu pro řešení puzzle. Idea je následující. Často se stává, že shluk políček je ohraničen právě dvěma barvami (je jimi uzavřen), mezi kterými není prozatím žádný vztah. Pokud by tyto dvě barvy byly shodné, tedy byly by ve stejné barevné komponentě, potom by kolem shluku políček vznikl barevný cyklus. Jednoduchým testem lze zjistit, zda je tento cyklus přípustný, tj. zda nevynutí krátkou smyčku. Pokud by ji tvořil, potom musíme oddělit tyto dvě barvy – musejí být navzájem opačné. Tento jev se objevuje relativně často a umožňuje tak řešit i oblasti, kde se nevyskytuje příliš určených hran. Bohužel mě nenapadl žádný (časově) efektivní algoritmus, který by tuto myšlenku implementoval. Uvažoval jsem o modifikaci sweep-line algoritmu (tj. z pohledu topologie), či o 3-souvislých komponentách (tj. z pohledu teorie grafů), přesto jsem k efektivnímu řešení nedošel.

7.2 OBECNÉ TVARY POLÍČEK

Přestože nejvíce oblíbenou variantou pro hráče je stále standardní čtvercový tvar políček, mnohé programy (např. LoopDeLoop [5] a Loopy [6]) přicházejí s implementací obecného tvaru políček. Políčka tak mohou být například trojúhelníky, pětiúhelníky, šestiúhelníky, osmiúhelníky nebo různé kombinace předchozích.

Program byl tvořen výhradně pro standardní tvar políček. Nicméně většina metod by byla prakticky použitelná i pro jiné tvary – zejména dedukce cest, obarvování hran, CAL zámky a další. Komplikace by nastaly u pravidel. Musel by se speciálně vygenerovat další set pravidel pro každý podporovaný rozměr políčka. Pro větší tvar políček (pětiúhelníky a větší) by to znamenalo jak větší velikost pravidel, tak i jejich počet by byl větší. I přes tyto obtíže by bylo možné program modifikovat tak, aby podporoval obecné tvary políček.



Obrázek 7.2 – netradiční puzzle (příklad z aplikace LoopDeLoop)

Na obrázku 7.2 je znázorněn netradiční tvar puzzle včetně jeho řešení. Jde o skloubení čtvercových a trojúhelníkových políček.

8 ZÁVĚR

Snažil jsem se vytvořit aplikaci, která by dosahovala velmi dobrého poměru mezi rychlostí vygenerování puzzle a jeho obtížností. Myslím, že tento záměr se mi povedlo zdárně uskutečnit. Cílem práce nebylo vytvořit aplikaci, která by řešila úlohu hrubou silou (backtrackingem), nýbrž nalézt nový přístup pro její řešení pomocí různých metod a pohledů na tento problém. Jako vedlejší produkt má tato aplikace tedy negativní stránku – neschopnost řešit veškerá puzzle, jelikož úloha je NP-úplná. Pozitivní stránkou je polynomiální složitost a tedy i garantovaný rozumný čas pro vygenerování velkých puzzle.

Podstatným základem, umožňující splnit výše vytyčený cíl, bylo navržení CAL zámků, které mají daleko větší schopnosti než klasické zámky. Dále neméně důležitou roli tvoří navržení struktury pravidel políček a mřížových bodů. Jelikož jsou zakódována do jediného čísla, umožňují velmi rychlou aplikaci – stačí udělat logický součin. Podstatné je rovněž použití front, abychom dedukovali pouze tam, kde je to opravdu zapotřebí. Další důležitou část tvoří dedukce krátkých cyklů a cest. V žádné jiné aplikaci, která by nevyužívala přímo backtracking, jsem se s tímto přístupem nesetkal. Umožňuje to dělat řadu dedukcí a učinit tak pokrok při řešení puzzle. O propagaci do okolí mimo pravidel přispívá obarvování políček a hran.

Postupný vývoj aplikace (a reakce na puzzle jím vygenerovaných) lze nalézt na fóru kwontomloop.com [7], kde je založen thread s názvem New Puzzle Generátor. Tímto se také předem omlouvám za mou angličtinu, jelikož příspěvky byly psány narychlo.

Porovnání s článkem, ze kterého jsem původně vycházel

Článek od Stefana Hertinga z univerzity v Kentu [3] se zabývá řešením puzzle pomocí vygenerovaného setu pravidel o rozměrech 2x2 políčka. Na konci článku je uveden odkaz na puzzle o rozměrech 45x30 [10, puzzle 34], s vyšší obtížností. Autor článku zde uvádí, že při použití pouze backtrackingu nenašel řešení ani po 3 hodinách a program ukončil. S použitím jeho připravené množiny pravidel potom dokázal určit 1754 hran z celkového počtu 2775 hran, a to v čase 156 sekund. SL Pro dokáže vyřešit toto puzzle celé (úroveň expert a použití obarvování políček) a to v čase 0,4 sekundy.

Porovnání s ostatním SW pro tvorbu Slitherlinku

Má aplikace Slitherlinku je v porovnávacích testech zastoupena názvem SL Pro. Pro testy byl použit notebook s procesorem Intel Core2Duo 2GHz a se 3GB RAM. Testy byly prováděny opakovaně pro získání lepších výsledků.

- ***Porovnání s aplikací Loopy***

Tato aplikace [6] má tři úrovně – Easy, Normal, Hard. Taktéž nepoužívá backtracking a dosahuje tak rychlých časů pro generování puzzle. Je

implementována v C++. Jednoduchá úroveň je, co do obtížnosti, shodná s mou velmi jednoduchou úrovní. Úroveň Hard je pak plně pokryta mou pokročilou úrovní s obarvováním políček (dokáže toho více než Hard úroveň Loopy; u mne je to 3. úroveň obtížnosti ze 7). Loopy používá něco jako zámky a neimplementuje obarvování políček, ale jakési seskupování hran do komponent (tj. něco jako jednodušší verze obarvování hran). Pro zkušenějšího hráče je i obtížnost Hard relativně jednoduchá. Hlavní nevýhoda také spočívá v tom, že Loopy nepoužívá žádnou frontu událostí, ale snaží se řešit puzzle iterativně.

Pokud jde o porovnání výkonu aplikací, testujeme generování větších puzzle, kde se rozdíly více projeví. Budeme uvažovat puzzle o velikosti 40x30 a odpovídající si (přibližně - nelze definovat přesně) úrovně.

Loopy – Easy	12-13 sekund	SL Pro	5-7 sekund
Loopy – Hard	110-120 sekund	SL Pro	19-22 sekund

- ***Porovnání s aplikací LoopDeLoop***

Tato aplikace [5] implementuje zámky, obarvování hran a políček. Neimplementuje však žádné dedukce cest. Aplikace využívá i backtracking pro vygenerování těžších puzzle. Výhoda programu LoopDeLoop spočívá v tom, že umožňuje generovat puzzle i na jiných hracích plochách (pentagon, hexagon, oktagon, a jiné).

Poté, co jsem začal přispívat na fórum kwontomloop.com a uveřejňovat výsledky a puzzle, zaznamenala tato aplikace mnoho změn. Autor se snažil analyzovat má puzzle a navrhl několik nových metod, které mu umožnili generovat mnohem těžší puzzle bez backtrackingu, a to v lepším čase než měl dříve. Stále však je má aplikace rychlejší a má více dedukčních metod, které dokážou vygenerovat těžší puzzle.

Úroveň expert u SL Pro dokáže vygenerovat 40x30 puzzle v čase 90-190 sekund. LoopDeLoop dokáže vygenerovat stejně velké puzzle bez použití backtrackingu okolo 300 sekund (a je znatelně jednodušší).

Například puzzle (uvedené v Dodatku 10.1) bylo vygenerováno v čase 1,5 minuty. Programu LoopDeLoop trvalo více jak 25 minut vyřešit toto puzzle, a jak autor říká, jedna polovina hrací plochy musela být vyřešena pomocí úplné rekurze. Pravděpodobně je to způsobeno dedukcemi cest a pokročilými pravidly.

Na fóru kwontomloop.com jsem nalezl porovnání obtížnosti mezi SL Pro (výše uvedeným puzzle) a LoopDeLoop (několik puzzle bez backtrackingu vygenerované v dosud nejnovější dostupné verzi s využitím nových metod). Cituji „*Just solved veenca's latest beast - nice puzzle. Certainly tougher than any of the recent LDL set tilips posted*“.

- ***Porovnání s aplikací Slinker***

Tato aplikace [9] se liší od předchozích v přístupu k řešení puzzle. Vůbec neuvažuje zámky, ale má s sebou přiložený soubor pravidel. Pravidla jsou v následujícím tvaru. Je popsána situace na výřezu hrací plochy pomocí nápověd, křížků a čar (předpoklad) a poté seznam nově vydedukovaných hran (důsledek). Navíc jsou pravidla ohodnocena jistou obtížností a jsou ručně psána autorem. Z toho plynou ale i určité nevýhody. Jakékoliv puzzle s předem pevně daným setem pravidel je, po naučení těchto pravidel, snadné k vyřešení. Ačkoliv pravidla mají různou velikost, přesto nemohou postihnout spoustu pro uživatele zřejmých situací. Příkladem může být diagonální 322…. 223 pravidlo. V souboru pravidel lze nalézt i pravidla, která zabraňují krátkým cyklům (bez znalosti širšího okolí). Takováto pravidla nejsou nejlepším přístupem, jelikož zabraňují vyřešit puzzle malých rozměrů. Efektivita programu taktéž není závratná, jelikož se prochází vždy každá pozice na hrací ploše a pokouší se aplikovat každé pravidlo včetně jeho symetrií. Přesto jde o zajímavý přístup k problému a v kombinaci s použitím backtrackingu lze vygenerovat zajímavá puzzle. S rostoucím rozměrem hrací plochy však rapidně roste čas k vygenerování puzzle (i bez backtrackingu). Přímé porovnání s tímto programem nelze provést, jelikož zde není funkce pro vyřešení celého puzzle.

- ***Ostatní SW***

Ostatní software, který jsem nalezl, nebylo možno porovnávat. Jedním z důvodů bylo, že programy umožňovaly hráči puzzle pouze řešit, postrádaly generátor. Byla v nich pevně daná sada puzzle, a po jejich dohrání ztrácela aplikace smysl. Druhým důvodem bylo, že programy negenerovaly jednoznačná puzzle. Taková puzzle není těžké vygenerovat – stačí vygenerovat smyčku a náhodně odstranit skupinu nápověd. Puzzle však postrádají zábavnost.

Přiložený program

K bakalářské práci je také přiložen na CD program, implementující všechny navržené algoritmy. Program není třeba instalovat, ale ke svému chodu potřebuje Microsoft .NET Framework 3.5 SP1.

9 LITERATURA A ZDROJE

[1] Wikipedia - Slitherlink
http://en.wikipedia.org/wiki/Slitherlink

[2] Yato T. (2000): On the NP-completeness of the Slither Link Puzzle, IPSJ SiG Notes, AL-74:25–32, 2000
http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/SIGAL74-3.pdf

[3] Herting S. (2005): A rule-based approach to the puzzle of Slither Link
www.cs.kent.ac.uk/pubs/ug/2005/co620/slither/report.pdf

[4] Kráľ D., Mareš M., Straka M.: KSP, Recepty z programátorské kuchařky - Minimální kostra, DFU
ksp.mff.cuni.cz/tasks/16/cook3.html

[5] LoopDeLoop, aplikace
http://www.themissingdocs.net/wordpress/?p=43

[6] Loopy, aplikace, Simon Tatham's Portable Puzzle Collection
http://www.chiark.greenend.org.uk/~sgtatham/puzzles/

[7] Forum o slitherlinku
http://www.kwontomloop.com/forum.php

[8] Wikipedia - Fisher–Yates shuffle
http://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle

[9] Slinker, aplikace
http://code.google.com/p/slinker/

[10] Fujiwara H., kolekce Slitherlikových puzzle
http://www.pro.or.jp/ fuji/java/puzzle/numline/index-eng.html.

[11] Virius M. (2002): Od C++ k C#. Kopp

10 DODATKY

10.1 SL Pro puzzle

40x30:a2a2a3a2b3a3a3b1a2b2a3f2a3c3b22b2a1b1231b22a31f3a23a1c1c21a2a2b1e1d3a3a22b22b3
c3a3a12331a3a3d1f3a22b0b2222a2a1a2a13c2a2d0a31122a12d12b2b2b22e11d2d21b2e3b1a211c2a
1a2a31c3a23a21b221b0a2b1a2b11b3d32a3c2d2c3f3a3a1a1e13a3a1f1a1b203a21a22a1c222b3a1d1c
22c2d02c0a31a3c1a12a1a2b2a1a12b3b3b3d1a2a2d2a2c31b1b313b2a2a2a2a3h22b311b31b3a2313e
21f113h2a2a2d32e23a12a1b20a2c1a1b2e213a3c21a3c12a222a2a3a2c1a13a3b2a3a2b2f2a2b21c23c
3e1a3a2d2k12b2g1c32a3b21a12a1b1a3312a031b12a12a23b131a1b0b2f3c2a3f1a12b1a2b1c3a3d32
3c02a21b32a0a2a22b2b1a13e122b2a2b2g1d3a2d3a3c222a3a3b12f1a2c33d1a2d22a22c21b1a03b03
b233222b233210d1a2a3222b1g1d1e3b2c23a23g2211a33b0b2a21d1e2c0c2b1a2k3e212a02b31d3a3
13a3a0c1a2b1a1a2b11a2a1b3d2c2c1d1311032a30b2b12a2b3a2a2b2a12a1a1c1a12b2e222c2d3a2a1
a1a2b2a1a22a11a2d2a11a1a23c122c2b1232c32b2d2b2b2c2b3c

10.2 Demonstrace dedukcí CAL zámků