Spravedlivé hledání min

Program má oddělenou grafickou část a část zodpovědnou za správný běh hry. Druhá část je umístěna v adresáři service, ale její úplně hlavní částí je `MinesweeperService`. Zmíním-li tak nějakou funkci bez udání třídy, půjde vždy o tuto.

Nejprve krátce o grafickém rozhraní

Jako u každé Angular aplikace, grafická část sídlí v `app component`. V našem případě používá dvě další komponenty, `timer` a `square`, timer jsou dva číselníky a square jsou samotná herní políčka. Kolik políček hra má a co na nich má být zobrazeno však rozhoduje service, a proto pomocí `ngFor` iterujeme `columnArrayí`, ze které se berou samotná data pro políčka.

Myslím, že zazší popis grafické části není potřeba, protože zde není nic zvlášť chytrého nebo záludného, jde čistě o pokus napodobit původní hledání min. Všechny komponenty mají velikosti nastavené absolutně v pixelech, což věc mírně usnadnilo (nebylo potřeba pracovat s relativními velikostmi pro různé displeje). Ještě zmíním, že veškeré ikony ve hře jsem kreslil v programu Inkscape.

Spavedlnost

Nyní už k implementaci 'spravedlnosti', resp. zjišťování, která políčka jsou bezpečná, která určitě obsahují minu, a která jsou nejistá, nazvěme to *stavy* políček. Dále popsaný algoritmus je implementován metodou `evaluate`. Ta je z `AppComponent` vyvolávána po každém stisknutí volného políčka (kromě těch ukončujících celou hru).

K problému se dá jistě přistupovat různě. Nejsnazší přístup by byl použít omezené množství her, které by se střídaly a program by předem znal jejich vlastnosti. Popřípadě dokonce takových her, kde k nejednoznačnostem vůbec nedochází (pak by se výběr hry musel samozřejmě odvíjet od prvního tahu hráče). Ačkoli takový přístup zní jako podvádění, nebyl cizí ani originálním tvůrcům min [1]. Já se touto cestou ale nevydal a rozhodl jsem se problém řešit ve vší obecnosti a nechat hráče taky zvolit rozměry desky a počet min. I tak je ale jistě stále spousta způsobů, jak problém řešit a je možné, že jsem nezvolil ten nechytřejší. Pojďme rozebrat, jak jsem došel k dnešní podobě algoritmu.

O samotném problému řešení minesweeperu se dá dočíst [2], že je NP-úplný. V této problematice si ještě úplně nevěřím, tedy nechci tvrzení slepě věřit, a taky je možné, že znalost skutečného rozestavení (jako správce hry) problém redukuje na deterministicky polynomiální, ale ať je to jakkoli, polynomiální řešení se mi najít nepodařilo. Celý následující segment tak bude popisovat způsoby optimalizace algoritmu na přijatelně rychlý při praktickém užití.

[1] `http://www.minesweeper.info/wiki/Windows_Minesweeper#Basic_Facts`
[2] `http://web.mat.bham.ac.uk/R.W.Kaye/minesw/ordmsw.htm`

Brute force

Nabízí se začít s myšlenkou zkrátka iterovat všemi rozestaveními min, ověřit, která souhlasí s již zobrazenými čísly a u souhlasících ukládat pro každé políčko počty případů, ve kterých se na políčku mina nacházela. Pokud to nebude žádný případ, políčko je bezpečné, pokud všechny, na políčku určitě mina je. Je to samozřejmě velmi hloupý postup, ale bude se od něj odvíjet finální algorimus.

Narazili jsme taky na to, že se hodí vědět, která čísla už jsou na desce odhalena, resp. obecně se bude hodit nějaká datová struktura pro ukládání seznamů políček. Vyžadujeme od ní funkce vkládání, odebírání podle indexu a získání podle indexu. Účel splní třída `MSList`, není komplikovaná, obsahuje pouze dvě (nafukovací) pole s x-ovými a y-ovými souřadnicemi a informaci o délce seznamu. Odstranění na indexu provedeme čistě vložením posledního prvku na daný index a snížením uložené délky. Změna pořadí po odstranění prvku nám totiž nebude vadit.

Struktura je mimochodem použita už při náhodném rozmisťování min (metoda `setBombsRandomly`), kde vložíme všechna políčka do `MSListu`, a postupně vybíráme náhodné indexy, kde budou miny a ty ze seznamu odstraňujeme.

'Nasycená' čísla a dělení na 'komponenty'

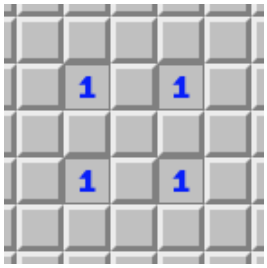
Testovat úplně všechna políčka na možnosti výskytu min po každém stisknutí je samozřejmě nesmysl, ať už proto, že stavy některých políček už známe z dřívějších běhů algoritmu, nebo proto, že jsou mimo zobrazená čísla a jejich stavy zkrátka nelze určit.

V `MinesweeperService` najdeme MSList ' `numList` ', který uchovává zobrazená čísla. První co uděláme je, že čísly proiterujeme a zjistíme, zda nesousedí právě s tolika políčky s potenciálními minami, jakou ukazují hodnotu, nebo naopak s právě tolika políčky s jistou minou. V obou případech můžeme číslo nazvat 'nasyceným', odpovídajícím způsobem nastavit stavy okolních políček, a číslo odstranit z numListu. Tento proces zajišťuje metoda `eliminateCertain` a iteruje se v ní numListem tak dlouho, dokud se dají najít nasycená čísla, tím se totiž nasycenými mohou stávat i další políčka. Tato část je ještě polynomiální, protože i kdybychom každou iterací 'eliminovali' pouze jedno číslo v numListu, celkem dojde k $\Theta(n^2)$ výpočtům (n značí délku numListu), ale čistě z praktického zkoušení algoritmu je vidět, že tento jednoduchý postup zejména u her s menší hustotou min odhalí stavy spousty, někdy i všech, políček.

Pokud numList neskončil prázdný, můžeme si všimnout, že některá políčka vzájemně neovlivňují svoje stavy. Můžeme tedy zbylá čísla a jejich přilehlá volná políčka rozdělit na více celků, říkejme jim 'komponenty'. Volným políčkům ve zdrojových kódech říkám '*idle*'. Komponenty nalezneme speciálním prohledáváním, kde střídáme číselná a idle políčka. To, že spolu dvě čísla sousedí totiž ještě neznamená, že jsou ve stejné komponentě. Totéž platí i pro idle políčka. Na následujícím obrázku vidíme dva příklady rozdělení na komponenty. Levá část ukazuje dvě různé komponenty se sousedícími čísly, pravá naopak nesousedící idle políčka sdílející komponentu díky dvojce, která je spojuje (oba případy by už vyřešila samotná eliminace, teď ale jde čistě o předvedení konceptu komponent).

O rozdělení na komponenty se stará metoda `divideIntoComponents`, která zvlášť ukládá čísla a idle pole komponent do `componentNums` a `componentIdles`, tj. k-tý index v componentNums obsahuje číselná pole k-té komponenty v podobě MSListu a k-tý index component Idles podobně její idle pole.

To už nám umožňuje použít již popsaný brute force algoritmus zvlášť na všechny komponenty. Má to ale háček. U původního brute force jsme mohli jednoduše iterovat všemi rozestaveními uživatelem zvoleného počtu min. U komponenty ale předem nevíme, kolik min bude obsahovat, dokonce může mít více platných rozestavení s různými počty min. Např. na následujícím obrázku vidíme komponentu, která může obsahovat jakýkoli počet min mezi jednou a čtyřmi.



Všimněme si, že metoda `evaluateSimple` s tím počítá. Jakmile v nějakém rozestavení min jsou čísla, která nemají okolo dost min, přičemž ostatní mají akorát, zkusí se i o jedna vyšší počet min. Symetricky se může zkoušet i nižší počet min. To, že se tak dojde ke všem 'legálním' (odpovídajícím zobrazeným číslům) rozestavením dané komponenty je triviální, protože se vždy zkouší všechna možná rozestavení min daného počtu, tedy i ty, které odpovídají legálním rozestavením s odebranými, resp. přidanými minami.

S jakým počtem min ale začít? Nevadilo by začít jakýmkoli číslem od nuly (komponenta sice vždy minu mít bude, jinak by byly její pole eliminována již v prvním kroce, nulový počet min však bude později potřeba u kompozitních komponent) do počtu idle polí komponenty. My ale konečně využijeme fakt, že známe skutečný počet min v komponentě, spočítáme je a na této hodnotě začneme. Ještě se nabízí otázka, proč rovnou nezkusit všech 2^n (n je počet idle polí komponenty) rozestavení min, čímž si asymptoticky nepohoršíme. Vzhledem k tomu, že se ale dá čekat malý rozsah legálních počtů komponent, jde stále o vítané urychlení.

Postup je neúplný

Bohužel, současný postup není dostatečný.

1. Pokud např. máme poslední dvě komponenty, zbývá-li otestovat posledních 5 min, první komponenta může mít mezi dvěma a čtyřmi minami a druhá má určitě právě tři miny,

znamená to, že první komponenta má právě dvě miny a rozestavení o třech a čtyřech minách nesmíme brát v úvahu.

2. Může se stát, že všechna neurčená políčka mimo komponenty buď jistě obsahují miny, nebo jsou určitě bezpečná, v závislosti na tom, kolik políček už má zjištěné stavy

První problém se nejsnáze vyřeší tak, že v podobných případech přejdeme k původnímu brute force postupu a všechna zbylá čísla i idle políčka bereme jako jednu velkou komponentu. Vydejme se ale jinou cestou. Legální rozestavení v komponentách nemusíme rovnou převádět na stavy, ale můžeme si je uložit a využít později, konkrétně v metodě `setStates`, kde se používají prefixové součty minim a maxim min v komponentách k zjištění limitů počtu jejich min. K ukládání těchto informací o komponentách použijeme abstraktní třídu `EvaluationData`. Jediné, co od ní očekáváme, je znát rozmezí legálních počtů min komponenty a funkce pro získání samotných stavů (resp. celkového počtu legálních rozestavení a ke každému poli počet legálních rozestavení s minou v tomto poli) pro zadané rozmezí počtu min. To se bude později hodit i k přesouvání bomb pro kýženou 'spravedlnost'. Druhý popsaný problém je jednodušší a je vyřešen v posledních dvou řádcích `setStates`.

Implementaci `EvaluationData` pro zatím popsaný algoritmus najdeme v `EvaluationDataSimple`. Pořád nás ale trápí exponenciální růst binomických koeficientů. Zkušenost konkrétně na mém zařízení říká, že řádově desetitisíce výpočtů provedené prohlížečem trvají ještě nepostřehnutelně krátkou dobu a statisíce už poznat jdou. Už $\binom{23}{11}$ je ale řádem v milionech, a i když tak velké komponenty nejspíš nebudou úplně běžné, musíme na ně být připraveni. Od toho je tu následující optimalizace.

Subkomponenty

Podívejme se teď na pozorování, které nám prozradí, že i komponenty se dají rozdělit na menší částečně nezávislé celky. Propůjčíme-li si k tomu ukázku z citované stránky o NP-úplnosti min, můžeme si všimnout, že komponenta jde dodatečně rozdělit na subkomponenty tak, že žádné číslo nesousedí naráz s idle polem dvou různých subkomponent. Konkrétně na následujícím obrázku vidíme zeleně vyznačenou jednu subkomponentu a oranžově druhou. Čísla taky jednoznačně připadají jedné či druhé komponentě podle toho, se kterou barvou sousedí. Navíc ale potřebujeme ještě červená pole, která sousedí s čísly obou subkomponent. Ve zdrojových kódech taková pole označuji jako 'mostová'.

Komponenty takto rozložené na části nazývejme *kompozitní*.

Jak nám toto rozdělení pomůže? Pokud bychom např. právě tuto komponentu zpracovali původním způsobem, testovali bychom $\binom{20}{8}$, tzn. 125′970 rozložení min. Po rozdělení ale

můžeme nejprve pro všechna rozložení min v mostových polích zpracovat obě subkomponenty jako standardní komponenty a poté, opět pro každé rozložení v mostech, zaznamenat jen ty 'legální'. To znamená (dvě komponenty, 8 idle polí komponenty - maximální binomický koeficient, 4 mostová pole) zhruba $2.\binom{8}{4}.2^4 = 2'240$ výpočtů (zanedbávám případné zkoušení různých počtů min v subkomponentách), tedy posunuli jsme se ze stovek tisíc výpočtů na tisíce.

Nějaký čas ale zabere i rozdělení na subkomponenty. Rozklad grafů by si nejspíš zasloužil vlastní zápočtovou práci, nebo i víc než to, já jsem se ale rozhodl pro následující jednoduchý algoritmus. Předně, 'malé' komponenty vůbec rozkládat nebudeme (nechť to jsou komponenty o méně než jedenácti idle polích).

Mějme uložena číselná pole subkomponent, tj. ze začátku prázdné pole MSListů. Nyní iterujme číselnými políčky původní komponenty. Iterace bude probíhat takto: je-li současné pole již součástí *dostatečně velké* subkomponenty, nedělej nic. Je-li ale subkomponenta *malá*, přidej do ní taky sousední čísla (sousednost skrze idle pole). Pokud současné pole není součástí subkomponenty, vytvoří se nová a vloží se do ní současné pole a všechny sousední pole. Nechť subkomponenta je '*malá*', pokud $n < \frac{5n}{m} + 1$, kde n značí počet číselných a m počet idle polí (heuristicky nastavená velikost).

Tím jsme získali několik seznamů číselných polí, ze kterých se dají přímo odvodit i idle pole subkomponent. Mohlo však dojít k následujícím nežádoucím jevům:

- některé subkomponenty by takto neměly vlastní idle pole - takové subkomponenty připojíme k sousedním (stále může dojít ke druhému problému)
- máme pouze jedinou subkomponentu - v takovém případě přistupujme ke každému číslu komponenty jako ke zvláštní subkomponentě a opravme případné výskyty prvního problému

Pokud i teď máme jedinou subkomponentu, rozklad jednoduše vzdáme (předpokládám, že tento jev bude vzácný).

Stejné komponenty dokola

Může být iritující, že popsaný postup provádíme po zcela každém hráčově tahu. Některé komponenty pravděpodobně evaluujeme vícekrát dokola. Všimněme si však, že typicky evaluace přináší několik polí s jistým stavem, což komponenty mění a i samotná kliknutí ovlivňují, do kterých komponent různá pole náleží. Ověřování opětovného výskytu stejných komponent by sice bylo polynomiální, to ale v praxi nemusí moc znamenat, vzhledem k tomu, že jsou i rozměry herní desky přeci jen omezeny. Rozhodl jsem se tak obětovat potenciální mírné zrychlení některých případů na úkor jiných a tuto 'optimalizaci' vůbec neimplementovat.

Přesouvání min a spravedlnost

Stavy polí už zjistit umíme. Teď naimplementujme samotnou spravedlnost. Popíšu pouze 'mazání' miny z nejistého pole při žádných bezpečných polích, Případ přidávání miny je symetrický.

Samozřejmě se budeme zabývat jen ovlivněnou komponentou, tedy po stisknutí pole s minou minu odstraníme a budeme předstírat, že jde o prázdné pole, přičemž prohledáme okolí pro jednotnou komponentu, na které budeme operovat. Pro jednoduchost (a zamezení zbytečnému zpomalení programu) nebudeme nastavovat žádná omezení pro počty min v komponentě. Dále už je přesouvání min analogické samotnému algoritmu pro zjišťování stavu polí v komponentě. Jednoduše se zjistí celkový počet možných rozložení min a vybere se z nich náhodné. Při tom se samozřejmě může změnit celkový počet min v poli, což vykompenzujeme čistě políčky mimo všechny komponenty. To sice stačit nemusí, potřebovali bychom měnit počty min i v ostatních komponentách, ale to by hru zbytečně zpomalilo, proto jsem se rozhodl ponechat nepatrnou možnost celkové změny min v poli, která se projeví na levém číselníku.

Výběr náhodného rozložení min je samozřejmě složitější u kompozitních komponent. Pro určité rozložení min v mostech je celkový počet legálních rozložení součin počtů možností v subkomponentách. Pro výběr správných rozložení v nich máme metodu `getSubiterations`, která de facto vybrané pořadí rozložení převede na číslo o tolika cifrách, kolik je subkomponent, kde každá cifra má váhu podle počtu legálních rozložení min v příslušné subkomponentě (což odpovídá zvolenému rozložení min).

Závěr

Zdá se, že program je teď relativně dost rychlý pro různé rozměry herního pole i pro naschvál nešikovně zvolená stisknutá pole. Myslím, že graficky je relativně věrnou imitací Windows Minesweeperu. Pokud se vám na hře, její implementaci nebo stylu mého kódu cokoli nelíbí, dejte mi prosím vědět. Jakákoli zpětná vazba se hodí. Pokud chcete algoritmus lépe pozorovat při práci, v nastavení hry je přístupná možnost zobrazovat stavy polí a můžete si tak hru nechat vyřešit za sebe.