Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Automatické generovanie bludísk a podzemných priestorov

Bakalárska práca

2018
Jozef Horváth

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Automatické generovanie bludísk a podzemných priestorov

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	RNDr. Michal Forišek, PhD.

Bratislava, 2018
Jozef Horváth

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

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

Názov: Automatické generovanie bludísk a podzemí
Automatic generation of mazes and dungeons

Anotácia: Práca sa zaoberá automatickým generovaním náhodných bludísk a náhodných podzemných priestorov s dôrazom pre použiteľnosť v stolných hrách.

Cieľ: Práca má nasledovné ciele:
1. Spraviť prehľad základných algoritmov na generovanie náhodných bludísk a podzemí.
2. Zvoliť si a implementovať niektorý z vyššie popísaných algoritmov, prípadne jeho vlastnú úpravu.
3. Do aplikácie navrhnúť a implementovať ďalšie vylepšenia smerujúce k lepšej použiteľnosti výstupu algoritmu pri niektorej konkrétnej stolnej hre (napr. Dungeons and Dragons).
4. Pokúsiť sa o jednoduché algoritmické vylepšenia implementovaného algoritmu (napr. generovanie tajných chodieb medzi miestnosťami).

Vedúci: RNDr. Michal Forišek, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Dátum zadania: 05.11.2017

Dátum schválenia: 08.11.2017 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Abstrakt

Výsledkom mojej bakalárskej práce je sprehľadnenie a popísanie dvoch typov algoritmov, následná implementácia, vlastné návrhy a vylepšenia. Venoval som sa algoritmom na generovanie náhodných bludísk a algoritmom na generovanie podzemí. Podzemiami sú myslené priestory ako jaskyne, kobky, mestá fantastických tvorov žijúcich pod zemou (napríklad trpaslíci) a iné. Tieto dva typy algoritmov sa občas zlievajú pri algoritmoch na generovanie podzemí. Následne som vytvoril vlastnú implementáciu generátora podzemí, ktorá generuje náhodné bludiská s miestnosťami vhodné pre rôzne hry. K prvotnej implementácii som potom navrhol a doplnil vylepšenia pre stolné hry, presnejšie pre Dungeons and Dragons. Prvým vylepšením som pridal rôzne typy dverí namiesto prázdneho prechodu. Druhé vylepšenie upravuje generovanú mapu aby používala zaužívané obrázky pre pridané typy dverí.

Kľúčové slová: bludisko, podzemie, implementácia

Abstract

Result of my Bachelor thesis is clarification and summary of two types of algorithms, followed by implementation and my own suggestions and improvements. I looked into algorithms for generating randomized mazes and algorithms for generating dungeons. By dungeons I mean caves, cell blocks, cities of fantastic creatures living underground (dwarfs for example) and others. These two types of algorithms sometimes coalesce in algorithms for generating dungeons. Subsequently I created my own implementation of dungeon generator, which generates randomized mazes with rooms appropriate for games. To this first version I suggested and implemented improvements for usage in tabletop games, Dungeons and Dragons to be precise. First improvement adds different types of door instead of empty passage. Second improvement makes the program generate a map using common images of the added types of door.

Keywords: maze, dungeon, implementation

Obsah

Zoznam obrázkov

Zoznam tabuliek

Úvod

Nižšie spomínam perfektné bludiská, na ich popis potrebujem termíny článok, cyklus a cesta. Článok je najmenšia časť bludiska alebo podzemia. Cesta je postupnosť článkov, cez ktoré musíme prejsť aby sme sa dostali z jej prvého článku po posledný článok. Cyklus je najmenší súbor článkov, ktoré obkolesujú aspoň jeden článok. Perfektné bludisko je typ bludiska, ktoré neobsahuje žiadne cykly. V takomto bludisku sa dá dostať z každého článku do každého iného článku bludiska a to práve jednou cestou.

V prvej kapitole sa venujem bludiskám. Najprv popisujem čo sú bludiská a uvádzam príklady ich využitia a výskytu. Bludiská môžu mať rôzne vlastnosti: dimenzia, hyperdimenzia, topológia, teselácia, trasa, štruktúra, zameranie a iné. Každá z týchto vlastností má svoje podvlastnosti, ktoré popisujú vlastnosti bludísk. Tieto vlastnosti popisujem v prvej sekcii. Niektoré z nich nerozoberám príliš do hĺbky, pretože nie sú pre túto práce relevantné.

V ďalšej sekcii popisujem samotné algoritmy na generovanie bludísk. Ku každému uvádzam verziu na tvorbu perfektného bludiska a dve ukážky bludiska. Algoritmy, ktoré popisujem sú: rekurzívny návratový algoritmus, Ellerov algoritmus, Kruskalov randomizovaný algoritmus, Primov algoritmus, rekurzívne delenie, Aldous–Broderov algoritmus, Wilsonov algoritmus, vyhľadávací algoritmus a rastový algoritmus.

V druhej kapitole sa venujem podzemiam. Keďže v slovenčine je podzemie najbližšie slovo k anglickému „dungeon“, používam toto pomenovanie. V úvode popisujem čo znamená toto slovo v zmysle tejto práce a pôvod tohto významu.

V jedinej sekcii tejto kapitoly píšem o algoritmoch na generovanie podzemí. Našiel som rôzne spôsoby ako generovať podzemia. Väčšina z nich využíva miestnosti, no niektoré tvoria podzemie len z chodieb. Popísané algoritmy sú: algoritmus náhodnej prechádzky, jednoduchý smerový algoritmus, podzemie vytvorené z miestností (dva spôsoby), mriežkový generátor podzemí, generátor podzemí v hre Angband, binárne deliaci generátor podzemí a podzemia s chodbami tvoriacimi bludisko.

V tretej kapitole popisujem svoj vlastný postup pri implementácii algoritmu „podzemia s chodbami tvoriacimi bludisko“ využívajúc rastový algoritmus na tvorbu chodieb. Program je písaný v jazyku Python 3 s využitím balíčku Wand.

V prvých dvoch sekciách zdôvodňujem svoj výber algoritmov, popisujem úpravy, vzniknuté problémy a moje riešenie k nim.

Nakoniec podrobne opisujem samotný program rozdelený na päť sekcií: importované funkcie a konštanty, pomocné funkcie, hlavné funkcie, trieda *cell* a hlavný program.

Kapitola 1

Terminológia

V texte používam termíny článok, cesta, cyklus, slepý koniec a slepá chodba. Ich význam je intuitívny, no aby som zabránil nedorozumeniam definujem tieto pomenovania nasledovne.

Článok je najmenšia časť bludiska alebo podzemia.

Cesta spájajúca články A a B je postupnosť článkov, ktorej prvým článkom je A, posledným B a každý článok postupnosti má prechod k predchádzajúcemu a nasledujúcemi článku. Pre článok A stačí ak má prepojenie na nasledujúci článok a článku B stači ak má prepojenie na predchádzajúci článok.

Cyklus je najmenší súbor článkov, ktoré obkolesujú aspoň jeden článok.

Slepý koniec je článok, z ktorého sa dá odísť len jedným smerom.

Slepá chodba je cesta začínajúca v slepom konci, z ktorej sa až po jej posledný článok nedá odbočiť iným smerom ako k predchádzajúcemu alebo nasledujúcemu článku.

Kapitola 2

Bludiská

Bludisko je typ logickej hádanky. Jeho cieľom je nájsť cestu z bodu A do bodu B. Popri správnom riešení sa oddeľujú cesty, ktoré zavádzajú riešiteľa. Bludiská sa využívajú ako forma rekreácie, no aj ako laboratórny test napríklad na skúmanie správania a schopností učenia sa zvierat, najmä myší a potkanov[19].

Špeciálnou verziou bludiska je labyrint. *Labyrint* je bludisko tvorené jednou pokrútenou cestou, tiež ho označujeme ako jednosmerné bludisko. Labyrinty sú ľudstvu známe už vyše 4000 rokov, kedy boli stavané alebo zobrazované na steny a podlahy. Ich pôvodný zámer síce nie je známy, no postupom času sme si pre ne našli rôzne významy: cesta k spáse, správny spôsob života, duchovné cvičenie, obraz zložitosti života, ...

Pravdepodobne najznámejší labyrint je Krétsky labyrint z gréckej mytológie. V legende je labyrint popisovaný ako zložitá spleť ciest, ktorou Tézeus[1] našiel cestu len vďaka pomoci od Ariadne [2]. Symbol tohto labyrintu, vyobrazený na minciach z mesta Knossos, však zobrazuje jednoduchý labyrint (viď. obrázok 2.1).

V dnešnej dobe sú po svete roztrúsené labyrinty a bludiská najmä s rekreačným a estetickým zámerom. Medzi najznámejšie patrí „Ananásové bludisko“ na Havaji (viď. obrázok 2.2), labyrint pri vile Pisani v meste Stra v Taliansku (viď. obrázok 2.3), bludisko pri paláci Hampton v Londýne (viď. obrázok 2.4) a labyrint v katedrále v Chartres vo Francúzsku (viď. obrázok 2.5).

[1] hrdina z gréckej mytológie, syn boha Poseidóna

[2] dcéra krétskeho kráľa Minoa z gréckej mytológie

Obr. 2.1: Minca zobrazujúca jednoducý labyrint s jedinou cestou. Takéto mince boli nájdené pri vykopávkách mesta Knossos.

Obr. 2.2: Ananásové bludisko na Havaji

Obr. 2.3: Labyrint pri vile Pisani v meste Stra v Taliansku

Obr. 2.4: Bludisko pri paláci Hampton v Londýne

Obr. 2.5: Labyrint v katedrále v Chartres vo Francúzsku

2.1 Vlastnosti bludísk

Bludiská môžeme rozdeliť podľa viacerých vlastností. Ja používam delenie podľa Walter D. Pullena [23] s menšími úpravami kvôli čitateľnosti. Bludiská delíme podľa siedmych základných vlastností: dimenzia, hyperdimenzia, topológia, teselácia, trasa, štruktúra a zameranie.

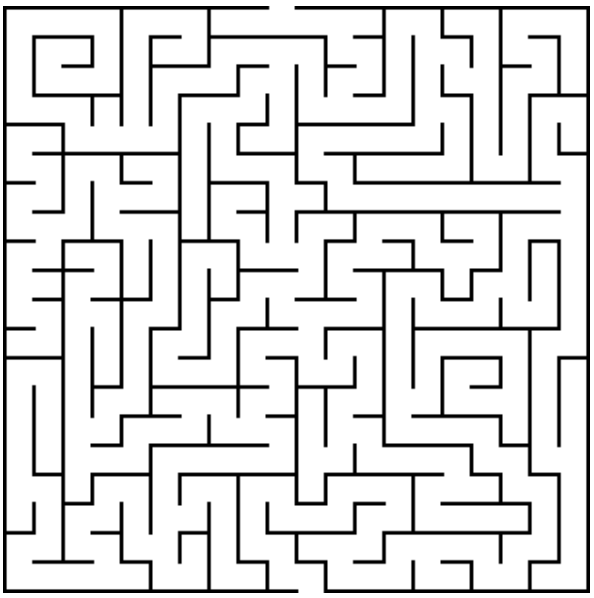
Dimenzia

Dimenzia bludiska hovorí koľkorozmerné je bludisko v priestore. Väčšina bludísk patrí do kategórie **2D** bludísk, tieto bludiská sa dajú nakresliť na papier bez toho aby sa cesty prekrývali (viď. obrázok 2.6).

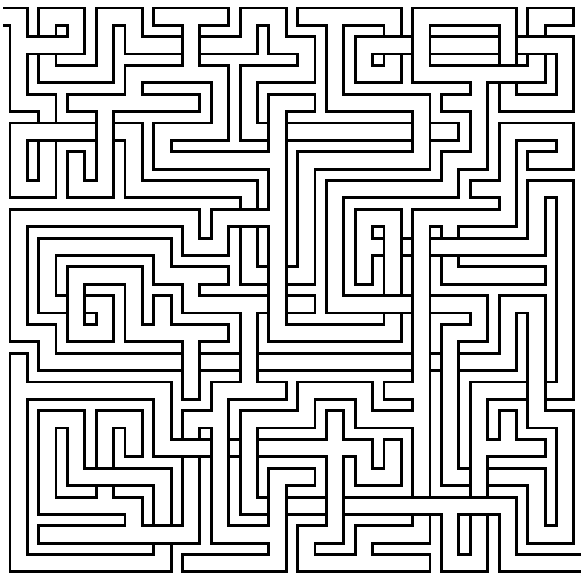
Bludiská tvorené viacerými prepojenými 2D bludiskami patria do kategórie **3D** bludísk. Samotné podlažia môžu byť prepojené napríklad schodiskami.

Vyššie dimenzie bludísk využívajú navyše špeciálne typy „portálov“ na cestovanie štvrtou dimenziou. Príkladom takýchto portálov je cestovanie do minulosti, kde prekážka, ktorá nám bráni v postupe, ešte nebola.

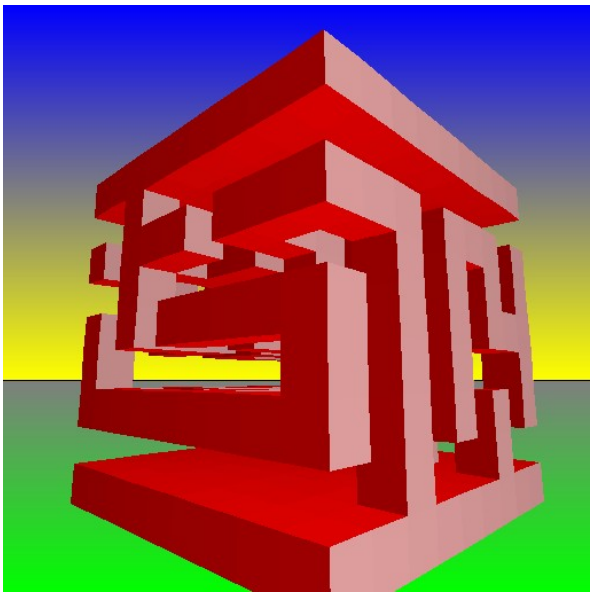
Poslednú kategóriu tvoria **2,5D** bludiská. Podobne ako 2D bludiská sa dajú nakresliť na papier, no navyše obsahujú cesty, ktoré idú popod iné cesty (viď. obrázok 2.7).



Obr. 2.6: Štandardné ortogonálne bludisko v 2D priestore.



Obr. 2.7: Bludisko s 2,5D dimenziou.



Obr. 2.8: 3D bludisko s hyperdimenziou prvého rádu. Bludisko na obrázku má viacero možných vstupov, pri každom z nich sa bludisko správa inak.

Obr. 2.9: 2D bludisko na povrchu 3D objektu, v tomto prípade na modeli ruky.

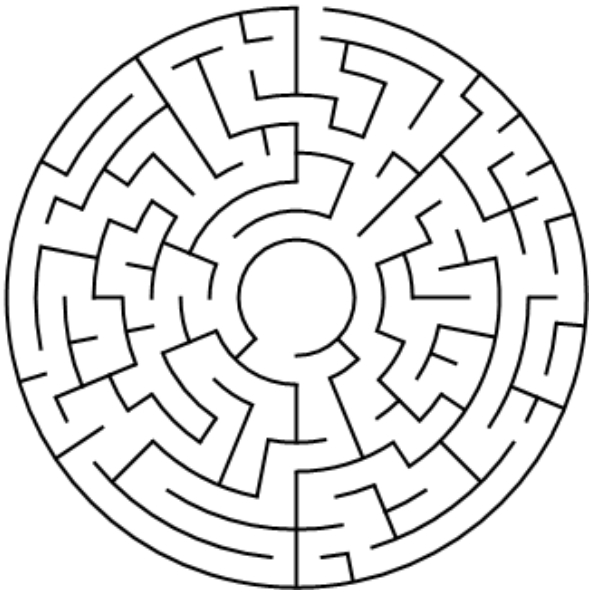
Hyperdimenzia

Hyperdimenzia definuje dimenziu objektu, ktorým prechádzame cez bludisko. Pre moju prácu je dôležitá len hyperdimenzia nultého rádu, no pre predstavu uvádzam príklad aj pre hyperdimenziu prvého rádu a popisujem spôsob akým spoznať bludiská vyšších hyperdimzií.

Hyperdimenzia **nultého rádu** hovorí, že bludisko riešime použitím bodu, alebo malého objektu. Riešením bludiska je čiara, ktorá spája začiatočný a koncový bod.

Bludiská s hyperdimenziou **prvého rádu** musia byť aspoň 3D (viď. obrázok 2.8). Objekt používaný na prejdenie bludiskom je čiara, napr. kúsok vlákna. Bludisko samotné je rozdelené na dve časti a cieľom je previesť čiaru pomedzi tieto dve časti. Počas prechodu bludiskom musíme čiaru vhodne ohýbať.

Bludiská s hyperdimenziou n–tého rádu používajú objekt s dimenziou (n-1) v bludisku dimenzie aspoň (n+1).



Obr. 2.10: Bludisko tvorené sústrednými kružnicovými cestami.

Topológia

Topológia ohraničuje tvar priestoru, ktorý bludisko ako celok zaberá. V práci sa venujem štandardným bludiskám, no pre zaujímavosť spomínam aj bludiská s neštandardnou topológiou.

Medzi **neštandardné** bludiská patria bludiská, ktoré využívajú prepojenie hrán. Po vložení bludísk na povrch nejakého objektu prepojíme oba konce (viď. obrázok 2.9). Medzi neštandardné bludiská patria bludisko na Möbiovom liste alebo kocka ktorej steny tvoria navzájom prepojené bludiská.

Štandardné bludiská pokrývajú ostatné typy bludísk.

Teselácia

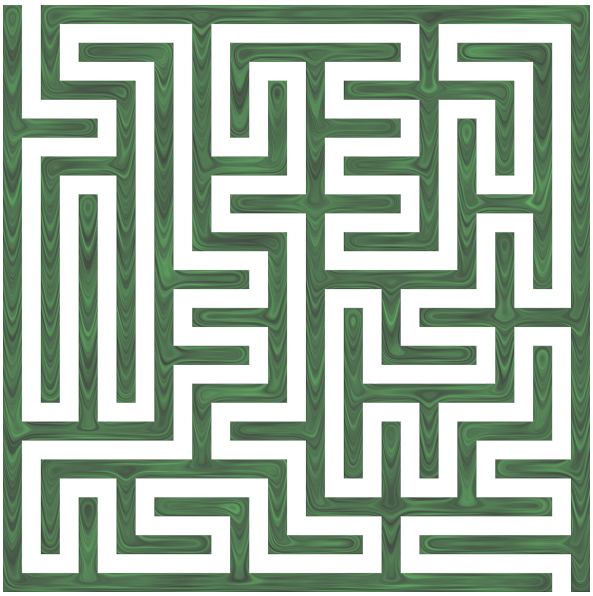
Teselácia popisuje tvar jednotlivých článkov tvoriacich bludisko. V tejto práci sa venujem len ortogonálnym bludiskám.

Ortogonálne bludisko tvorí štvorcová mriežka, v ktorej sa cesty pretínajú v pravom uhle. Ku každému článku sú pripojené maximálne štyri cesty.

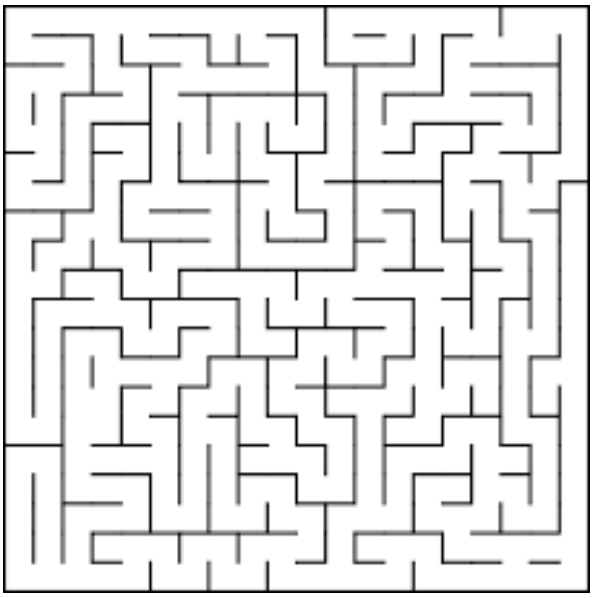
Medzi iné typy patria bludiská zložené z trojuholníkov, a iných tvarov alebo ortogonálne bludiská s uhlopriečnymi chodbami a iné. V prípade záujmu nájdete iné typy teselácie na stránke Think Labyrinth: Maze Algorithms. Na obrázku 2.10 môžeme vidieť príklad kruhového (Theta) bludiska.

Trasa

Trasa je zaujímavá vlastnosť ohľadom generovania bludísk. Limituje totiž výskyt cyklov a slepých chodieb v bludisku. Podľa trasy môžeme vytvoriť labyrint, perfektné,



Obr. 2.11: Bludisko tvorené jedinou točitou cestou známe ako labyrint.



Obr. 2.12: Cyklické bludisko nemá slepé uličky, no riešiteľ sa jednoducho dostane naspäť tam, kde začal.

cyklické alebo čiastočne cyklické bludiská.

Labyrint je bludisko bez križovatiek. Tvorí ho jedna dlhá pokrútená cesta (viď. obrázok 2.11). Vďaka tomu nemá žiadne slepé chodby ani cykly.

Perfektné bludisko neobsahuje žiadne cykly ani neprístupné oblasti. Z každého bodu existuje práve jedna cesta do každého iného bodu bludiska. Riešenie je teda len jedno.

Cyklické bludisko nemá slepé uličky, cesty sa stáčajú a znovu stretávajú. Toto vytvára v bludisku cykly a spôsobuje chodenie v kruhu (viď. obrázok 2.12). Dobre navrhnuté cyklické bludisko môže byť oveľa ťažšie než perfektné bludisko rovnakej veľkosti.

Čiastočne cyklické bludisko obsahuje slepé konce aj cykly. V tomto prípade možno cyklickosť čiastočne kvantifikovať, „veľmi zacyklené bludiská“ majú veľa cyklov, zatiaľ čo „málo zacyklené bludiská“ ich majú málo.

Štruktúra

Štruktúra je veľmi jemná vlastnosť, ktorá bližšie popisuje štýl prechodov. Štruktúra má priveľa premenných na to aby sme sa pozreli na všetky z nich, preto uvádzam len vychýlenie, beh, elitnosť, symetriu a jednotnosť.

Vychýlenosť do istého smeru hovorí, že prechody týmto smerom sú dlhšie ako prechody inými smermi. Napríklad horizontálne vychýlené bludiská majú dlhšie cesty v horizontálnych smeroch, zatiaľ čo vertikálne cesty sú kratšie.

Faktor **behu** bludiska poukazuje na dĺžku prechodov idúcich bez nutného odbočenia. Ak je tento faktor nízky, prechody sú krátke a bludisko celkovo vyzerá náhodne.

Faktor **elitnosti** popisuje dĺžku riešenia v porovnaní s veľkosťou bludiska. Elitné riešenie je často krátke a priame, no ťažko nájditeľné.

Symetrické bludisko má symetrické prechody. Časti bludiska môžu byť rotačne symetrické okolo stredu, symetrické podľa osi alebo bodu. Symetria bludiska môže byť čiastočná, úplná alebo môže opakovať istý vzor.

Jednotnosť je vlastnosť algoritmov na generovanie bludísk. Jednotný algoritmus vie vygenerovať všetky bludiská na istej veľkosti s rovnakou pravdepodobnosťou.

Zameranie

Zameranie nie je na bludisku viditeľné, rozdeľuje však spôsob tvorby bludísk na dve skupiny: pridávanie stien a tesanie chodieb. Toto síce väčšinou nemá vplyv na konečný výsledok, no má to vplyv na algoritmickú časť.

Pri **pridávaní stien** začína algoritmus s miestnosťou alebo ohraničením a postupne pridáva steny. Pri **tesaní chodieb** začína algoritmus s pevným kusom a postupne vyhlbuje cesty.

Niektoré bludiská môžu vzniknúť miešaním týchto spôsobov. Ak chceme v bludisku mať špirálu a na zvyšku nám nezáleží, dáme programu **predlohu**, ku ktorej následne dogeneruje bludisko.

Iné

Medzi viac–menej nezaraditeľné vlastnosti bludísk patria špeciálne vlastnosti ako smer prechodov a segmentácia. Niekedy môžeme chcieť aby sa niektorými prechodmi dalo prejsť len jedným **smerom**. **Segmentované** bludisko sa skladá z častí, ktoré majú rôzne vlastnosti v niektorých z vyššie uvedených kategórií.

Algoritmus	Slepé chodby (%)	Zameranie	Pamäť	Čas
Rekurzívny návratový	10	Chodby	N^2	27
Ellerov	28	Obe	$N*$	20
Kruskalov	30	Obe	N^2	33
Primov	30	Obe	N^2	160
Rekurzívne delenie	23	Steny	$N*$	10
Aldous–Broderov	29	Obe	0	279 (208)
Wilsonov	29	Obe	N^2	48 (25)
Vyhľadávací	11 (21)	Chodby	0	100 (143)
Rastový	49 (39)	Obe	N^2	48

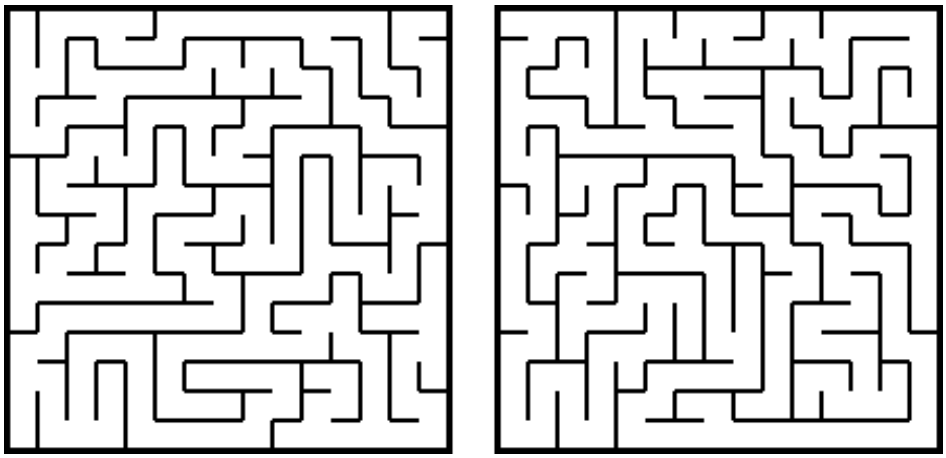
Tabuľka 2.1: Upravená tabuľka od Walter D. Pullena [23] popisuje vlastnosti perfektných bludísk vytvorených rôznymi algoritmami. Podľa zamerania by sme mali mať rovnaký výsledok, no ak sa výsledky líšia, v zátvorke je výsledok pre pridávanie stien. Slepé chodby je percentuálna časť bludiska, ktorú tvoria slepé chodby.
Zameranie hovorí, či algoritmus pridáva steny, teše chodby alebo môže používať oba prístupy.
Pamäť vyjadruje koľko pamäte navyše algoritmus potrebuje. Niektorým stačí mapa bludiska - 0, iné potrebujú pamäť veľkosti v závislosti od počtu článkov - N^2 a niektoré si ani nepotrebujú pamätať celé bludisko - $N*$.
Čas porovnáva ako dlho beží daný algoritmus v porovnaní s ostatnými, 10 je najrýchlejší algoritmus.
Riešenie je percentuálna časť bludiska, ktorá tvorí riešenie.

2.2 Algoritmy na tvorbu bludísk

V tejto časti popíšem algoritmy na generovanie perfektných ortogonálnych 2D bludísk. Keďže v perfektnom bludisku možno každé dva body spojiť práve jednou cestou, programy vytvárajú bludiská bez začiatočnej a koncovej pozície. Tieto si môžeme následne načítať ako vstup, po tom čo vykreslíme bludisko. V popise algoritmov tieto kroky neuvádzam.

Ak nie je povedané inak, stav článku predstavuje hondota v poli predstavujúcom bludisko. Táto hodnota je najprv 0. Po dokončení bludiska táto hodnota predstavuje súčet čísel, ktoré hovoria, ktorými smerom možno vyjsť z článku.

- (hore) H = 1
- (dole) D = 2
- (vpravo) P = 4
- (vľavo) L = 8



Obr. 2.13: Bludiská vytvorené rekurzívnym návratovým algoritmom

Pre všetky tieto algoritmy nám stačí vedieť rozmery bludiska, ktoré dostaneme ako vstup. Obrázky bludísk boli vygenerované na stránke o bludiskách, odkiaľ som aj čerpal informácie o týchto algoritmoch[8]. Niektoré z vlastností bludísk vytvorených týmito algoritmami sú popísané v tabuľke 2.1.

2.2.1 Rekurzívny návratový algoritmus [7]

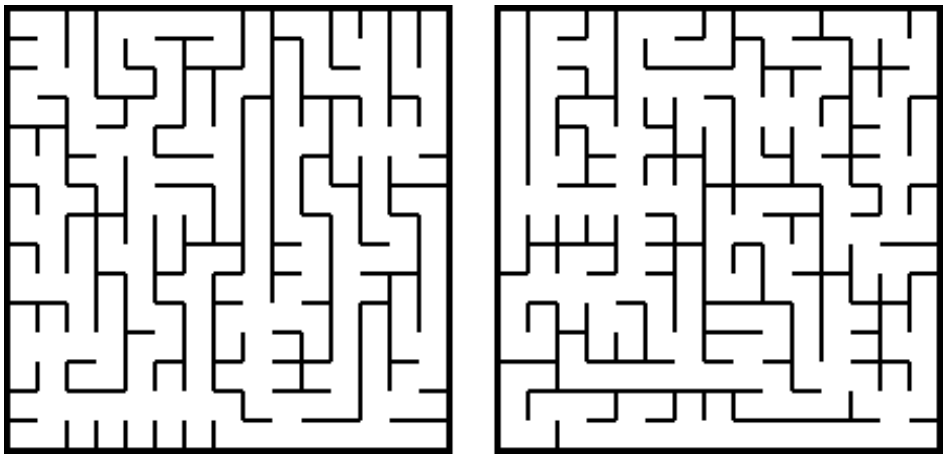
Tento algoritmus postupne tesá chodby, až kým nenavštívil všetky články. Ak nemá kam ísť, vráti sa k predchádzajúcemu článku. Takto skončí na článku ktorým začal.

Rekurzívny návratový algoritmus vieme aplikovať v dvoch krokoch. V prvom kroku si vytvoríme dvojrozmerné pole *plocha*, ktoré bude predstavovať články bludiska. V druhom kroku zavoláme rekurzívnu funkciu *kopTunel(0, 0, plocha)*.

Vo funkcii *kopTunel(cx, cy, grid)*[3] si najprv náhodne vyberieme poradie v akom sa pozrieme na okolité články. Toto poradie si pamätáme v lokálnom poli *smery*. Postupne prejdeme všetkými prvkami tohto poľa. Pre každý smer si nájdeme súradnice odpovedajúce článku daným smerom od aktuálneho článku (*nx, ny*). Ak je nový článok vhodný (t.j. nachádza sa medzi hranicami poľa a ešte nebol navštívený, čiže obsahuje hodnotu 0), pripočítame k hodnote starého článku príslušnú hodnotu smeru a k hodnote nového článku hodnotu opačného smeru. Potom zavoláme funkciu *kopTunel(nx, ny, grid)* so súradnicami nového článku. Po návrate z funkcie opakujeme s ďalším prvkom z poľa *smery*.

Dve ukážky bludísk vytvorených týmto algoritmom sú na obrázku 2.13.

[3] cx, cy sú súradnice aktuálneho článku v bludisku,
grid je dvojrozmerné pole bludiska



Obr. 2.14: Bludiská vytvorené Ellerovym algoritmom

2.2.2 Ellerov algoritmus[6]

Ellerov algoritmus je jeden z najzvláštnejších algoritmov na generovanie bludísk. Taktiež patrí medzi najrýchlejšie. Týmto algoritmom sa dokonca dajú generovať nekonečné bludiská v lineárnom čase. Keďže tvorí bludisko po riadkoch za použitia setov [4], nikdy sa nepotrebuje pozrieť na viac ako aktuálny riadok. Navyše vždy generuje perfektné bludisko.

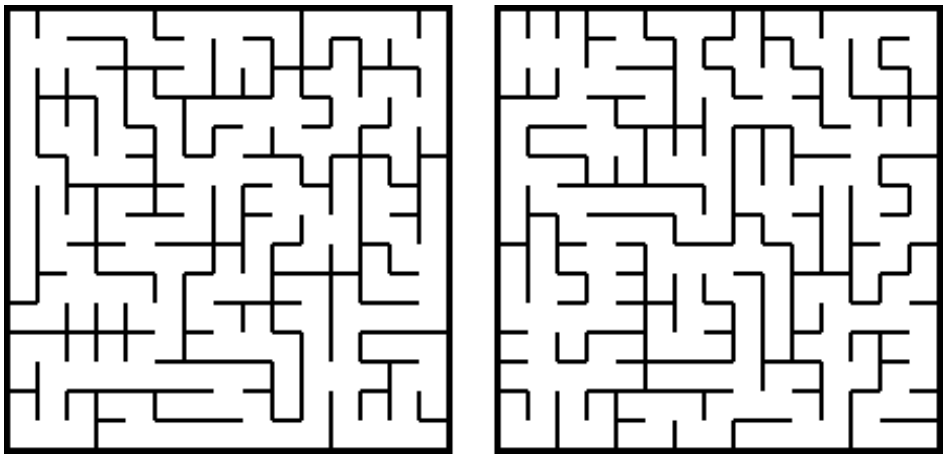
V aktuálnom riadku si najprv každý ešte neoznačený článok označíme ako samostatný set Potom náhodne odstraňujeme steny. Dávame si však pozor aby sme nespojili dva články, ktoré už majú rovnaký set. Následne pre každý set určíme aspoň jeden prechod do ďaľšieho riadku. Keď sa dostaneme do posleného riadku, prepojíme všetky rôzne sety.

Vieme, že dva články medzi sebou už majú nejakú cestu, ak patria do rovnakého setu. Naopak, aby sme zabezpečili, že z každého článku sa dá dostať do každého iného článku, musíme mať na konci iba jeden set. Preto v poslednom riadku spojíme všetky rôzne sety. Aby sme ich mohli spojiť, každý set sa šíri do ďaľšieho riadku. Keby sme tak neurobili, vytvorili by sme v bludisku izolovaný úsek bludiska.

Kľúčom k implementácii tohto algoritmu je implementácia setov. Potrebujeme rýchlo zistiť set článku a taktiež zoznam článkov v sete. To je možné udržiavaním hashu polí, ktoré mapujú sety článkov a druhého hashu, ktorý mapuje články k setom. Na identifikáciu setov nám stačí Integer.

Dve ukážky bludísk vytvorených týmto algoritmom sú na obrázku 2.14.

[4] *set* je skupina rôznych prvkov (prvok sa nesmie opakovať), poradie týchto prvkov nie je dôležité



Obr. 2.15: Bludiská vytvorené Kruskalovym algoritmom

2.2.3 Kruskalov randomizovaný algoritmus [12]

Kruskalov algoritmus je jeden zo spôsobov generovania minimálnej kostry z grafu s ohodnotenými hranami.

V randomizovanej forme ho možno využiť aj na generovanie bludísk. Všetky steny (možné prechody) si vložíme do jedného zoznamu. Zo zoznamu náhodne vyberieme stenu a ak to nespôsobí zacyklenie, odstránime ju z bludiska.

Podobne ako pri Ellerovom algoritme budeme používať sety, no ich implementácia bude iná. Aby sme mali rýchle spájanie setov, každý z nich bude predstavovať strom. Spojenie dvoch setov je potom napojenie jedného stromu ako podstrom druhého. Zisťovanie či dva články majú rovnaký set je porovnanie ich koreňov. Na reprezentáciu stien môžeme použiť jeden z jeho krajných článkov a smer prechodu.

Samotný algoritmus potom predstavuje cyklus, v ktorom postupne náhodne vyberáme steny, kým nevyprázdnime zoznam.

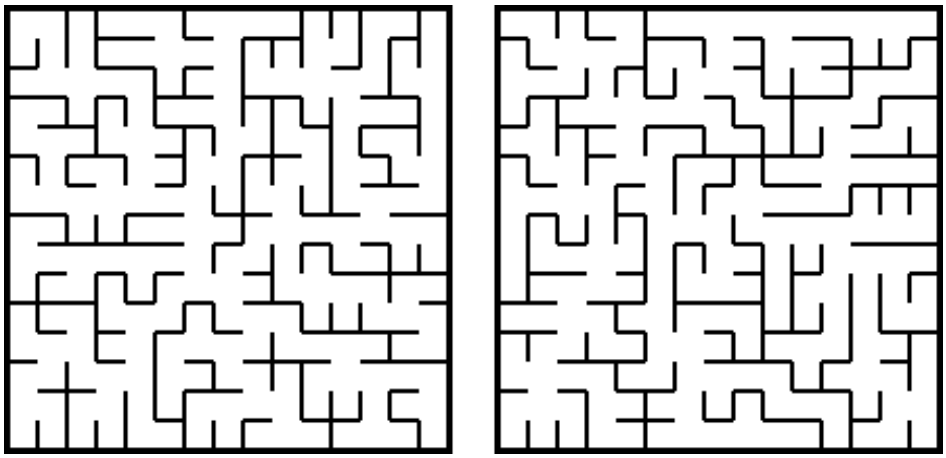
Dve ukážky bludísk vytvorených týmto algoritmom sú na obrázku 2.15.

2.2.4 Primov algoritmus[13]

Primov algoritmus podobne ako Kruskalov je jeden zo spôsobov generovania minimálnej kostry. Namiesto odstraňovania náhodných stien sa postupne rozrastá po článkoch.

Najprv si náhodne vyberieme článok, kde začneme. Tento článok pridáme do setu V, ktorý nám bude označovať články bludiska. Susedov tohto článku si pridáme do setu F, ktorý bude označovať články pripojiteľné k bludisku. Kým F obsahuje nejaké články, opakujeme tieto štyri kroky.

- Náhodne si vyberieme aktuálny článok z F.
- Susedov tohto článku, ktorí ešte nie sú v žiadnom sete, pridáme do setu F.



Obr. 2.16: Bludiská vytvorené Primovym algoritmom

- Náhodne vyberieme suseda aktuálneho článku, ktorý je zároveň v V a odstránime stenu medzi týmito dvomi článkami.
- Aktuálny článok presunieme zo setu F do setu V.

Dve ukážky bludísk vytvorených týmto algoritmom sú na obrázku 2.16.

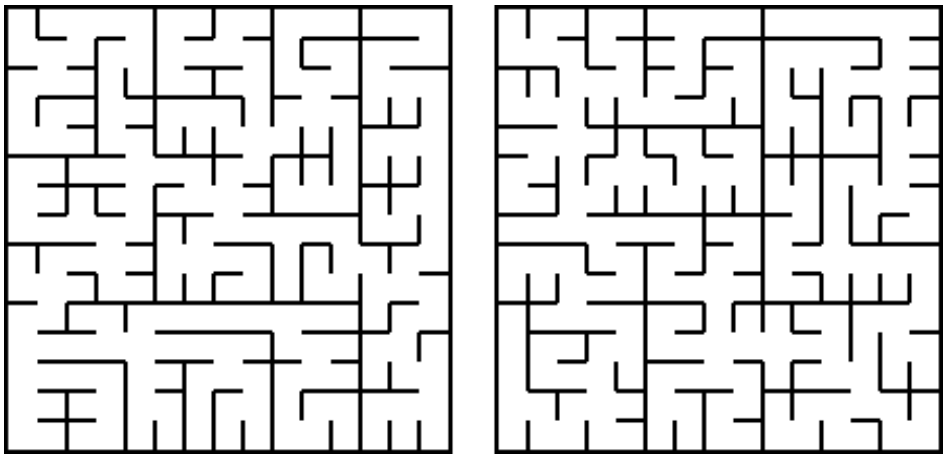
2.2.5 Rekurzívne delenie[14]

Tento algoritmus postupne pridáva stenu s jedným prechodom do plochy, čím ju rozdelí na dve časti a následne opakuje v oboch.

Budeme využívať jednu pomocnú funkciu *vyberOrientaciu(sirka, vyska)*. Táto funkcia rozhodne, že pridáme horizontálnu stenu, ak je *sirka* tejto sekcie menšia ako jej *vyska* a naopak. Ak sú obe rovnako veľké, vyberie náhodne. Potom zavoláme rekurzívnu funkciu *rozdel(plocha, x, y, sirka, vyska, orientacia)* na bode [0, 0] s orientáciou *vyberOrientaciu(sirkaBludiska, vyskaBludiska)*. Funkcia *rozdel* skontroluje, či ešte nemáme dokončený jeden rozmer. To znamená, že ak je *vyska* alebo *sirka* aktuálnej plochy menšia ako 2, tak sa môžeme vrátiť k predošlému volaniu.

Podľa orientácie si vyberieme buď x-ovú, alebo y-ovú súradnicu a náhodne vygenerujeme, kde bude stena. Potom zoberieme druhú z týchto súradníc a náhodne vygenerujeme, kde bude prechod cez túto stenu. Vložíme takúto stenu a určíme si okraje podplôch. Znovu zavoláme funkciu *rozdel* na oboch týchto podplochách v náhodnom poradí.

Dve ukážky bludísk vytvorených týmto algoritmom sú na obrázku 2.17.



Obr. 2.17: Bludiská vytvorené rekurzívnym delením

2.2.6 Aldous–Broderov algoritmus[9]

David Aldous a Andrei Zary Broder sú výskumníci, ktorí sa zaoberali jednotnými kostrami grafov a nezávisle od seba prišli s algoritmom na ich generovanie. [5] Tento algoritmus je jeden z najjednoduchších, no zároveň aj najmenej efektívnych. Na druhej strane generuje všetky bludiská danej velkosti s rovnakou pravdepodobnosťou.

Vyberieme si náhodný článok a označíme ho za navštívený. Potom prejdeme do niektorého jeho susedného článku. Ak tento nový článok ešte nebol navštívený, označíme ho ako navštívený a prepojíme s predchádzajúcim článkom. Opakujeme, kým nenavštívime všetky články.

Problém na ktorý pri tomto algoritme narážame je opätovné navštevovanie článkov. Keďže aj po navštívení už navštíveného článku pokračujeme z nového článku. Akákoľvek snaha o optimalizáciu by však mohla ovplyvniť schopnosť generovať všetky možné bludiská s rovnakou pravdepodobnosťou.

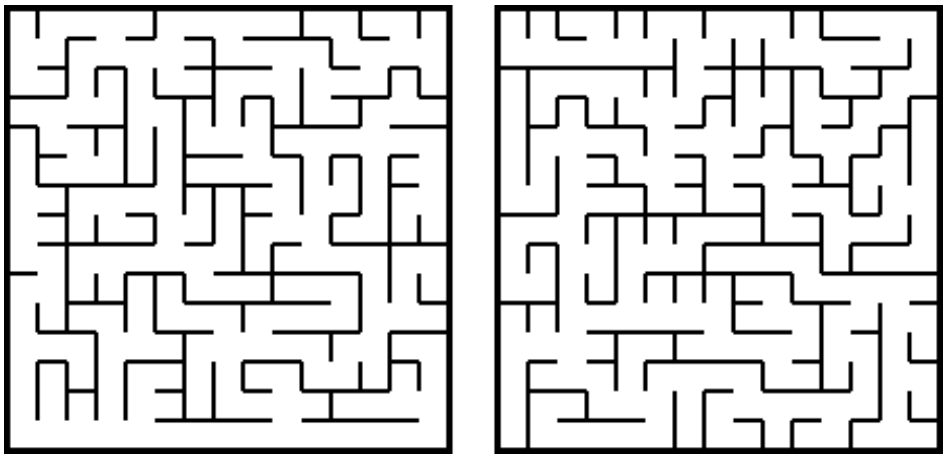
Dve ukážky bludísk vytvorených týmto algoritmom sú na obrázku 2.18.

2.2.7 Wilsonov algoritmus[15]

Wilsonov algoritmus tiež generuje jednotnú kostru grafu, no je pri tom o čosi efektívnejší.

Náhodne si vyberieme článok v poli a označíme ho ako časť bludiska. Kým existujú články vrámci rozmerov bludiska, ktoré ešte nie sú časťou bludiska, spustíme funkciu *prechadzka*. V tejto funkcii si vyberieme náhodný článok, ktorý ešte nie je časť bludiska a označíme si ho ako aktuálny článok. Zároveň si ho zapamätáme ako začiatočný článok pre aktuálnu iteráciu a vytvoríme si pole *navstivene*, kam si uložíme hodnotu 0 pre súradnice tohto článku. Potom v cykle opakujeme tieto kroky:

[5] Kostra grafu spája všetky body grafu. Jednotná kostra grafu je náhodne vybraná (s rovnakou pravdepodobnosťou) kostra zo všetkých možných kostier daného grafu.



Obr. 2.18: Bludiská vytvorené Aldous–Broderovym algoritmom

1. Náhodne si vyberieme smer z aktuálneho článku.
2. Skontrolujeme, či je článok daným smerom od aktuálneho článku v rozsahu bludiska. Ak nie je, vyberieme si iný smer.
3. Ak je, uložíme si smer, ktorým sme odišli z aktuálneho článku do poľa *navstivene* podľa súradníc aktuálneho článku.
4. Skontrolujeme, či je nový článok časťou bludiska. Ak nie, nový článok sa stáva aktuálnym a opakujeme cyklus od kroku 1.
5. Ak áno, ukončíme cyklus.

Keď ukončíme cyklus, vrátime sa k začiatočnému článku. Podľa smerov postupne pridávame články, ktorými sme prešli, do poľa (pridávame súradnice aj smer), ktoré potom vrátime. [6]

Po návrate z funkcie *prechadzka* postupne prejdeme vrátené pole a každý z jej prvkov pridáme do bludiska.

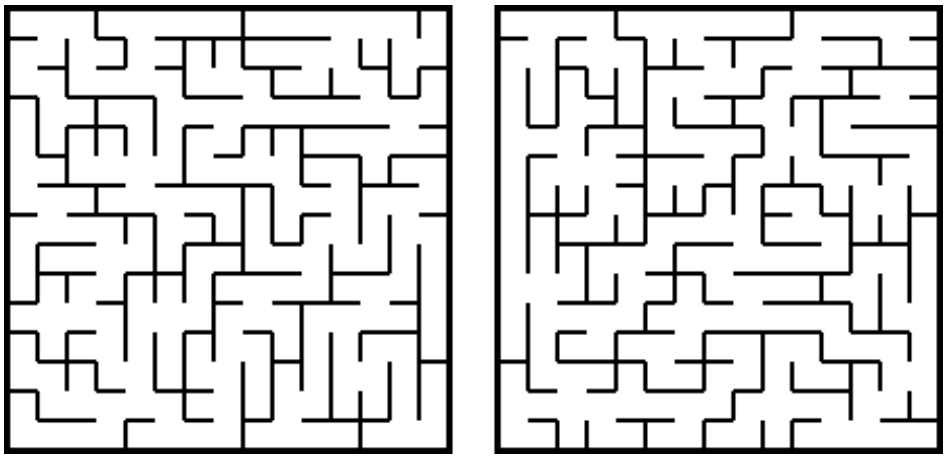
Dve ukážky bludísk vytvorených týmto algoritmom sú na obrázku 2.19.

2.2.8 Vyhľadávací algoritmus[11]

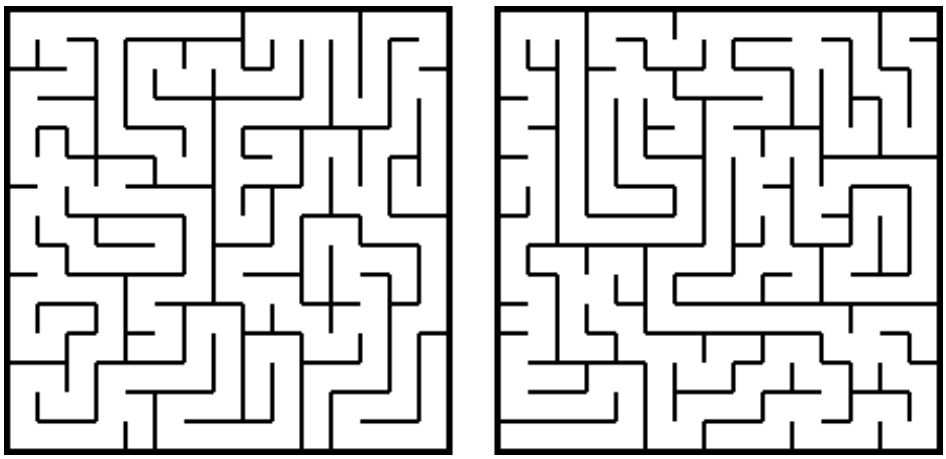
Náhodne si vyberieme aktuálny článok, v ktorom začneme tvoriť cestu bludiska. V cykle spúšťame funkcie *prechadzka* a *hladaj*.

Spustíme funkciu *prechadzka* so súradnicami aktuálneho článku. Vo funkcii *prechadzka* náhodne vyberieme vhodný smer (nový článok je vrámci okrajov bludiska a

[6] Týmto spôsobom nepridáme všetky navštívené články. Môže sa totiž stať, že niektorý článok sme navštívili viackrát. Pritom si pamätáme len posledný smer odchodu.



Obr. 2.19: Bludiská vytvorené Wilsonovym algoritmom



Obr. 2.20: Bludiská vytvorené vyhľadávacím algoritmom

ešte nebol navštívený) a prepíšeme oba články podľa potreby (k hodnote oboch pripočítame vhodnú hodnotu smeru). Následne vrátime súradnice nového článku. Ak z aktuálneho článku nenájdeme vhodný smer, vrátime nulu.

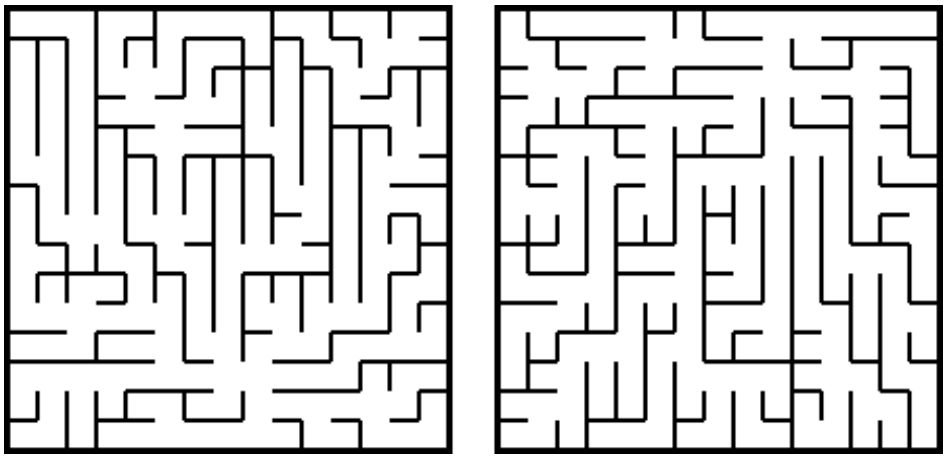
Ak sme z funkcie *prechadzka* vrátili nulu, tak spustíme funkciu *hladaj*. V tejto funkcii po riadkoch prejdeme celé pole. Hľadáme ešte nenavštívený článok, ktorého sused už bol navštívený. Obom pripočítame príslušnú hodnotu smeru. Vraciame súradnice tohto článku. Ak vo funkcii *hladaj* nenájdeme žiadny vhodný článok, bludisko je hotové.

Dve ukážky bludísk vytvorených týmto algoritmom sú na obrázku 2.20.

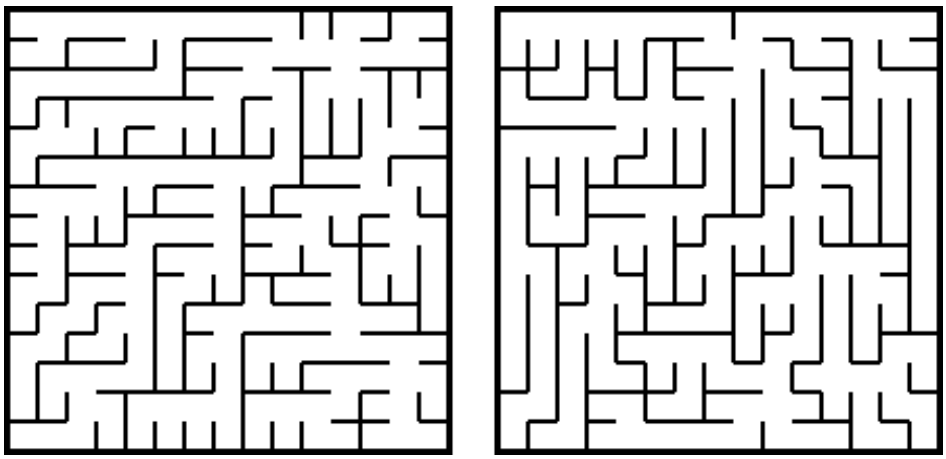
2.2.9 Rastový algoritmus[10]

Rastový algoritmus je pomerne jednoduchý na implementovanie. Počas generovania bludiska pracujeme so zoznamom článkov *clanky*. Pred spustením cyklu si doň pridáme náhodne vybraný článok. Kým pole *clanky* nie je prázdne beží cyklus:

1. Vyberieme prvok z *clanky*. Toto bude aktuálny článok.



Obr. 2.21: Bludiská vytvorené rastovým algoritmom s výberom náhodného článku.



Obr. 2.22: Bludiská vytvorené rastovým algoritmom s výberom najnovšieho článku

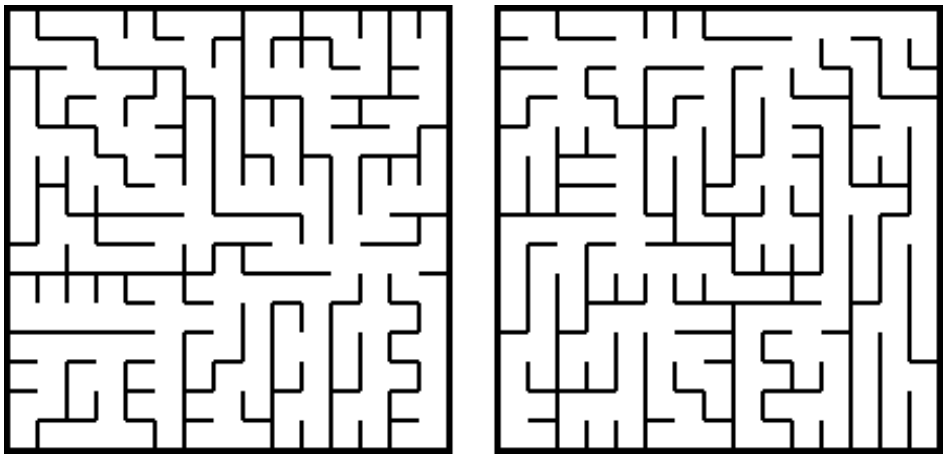
2. Náhodne zvolíme smer, ktorým pôjdeme z aktuálneho článku. Ak smer prekračuje hranice bludiska, alebo ukazuje na už navštívený článok, vyberieme iný smer. Ak žiadny smer nevyhovuje odstránime aktuálny článok z poľa *clanky* a vrátime sa k prvému kroku.

3. Aktuálny aj nový článok si pripočítajú hodnoty príslušných smerov.

4. Nový článok pridáme do poľa *clanky*.

Spôsob akým vyberáme prvky z poľa *clanky* ovplyvňuje, ako sa správa celý algoritmus. Ak budeme vyberať náhodný prvok, bude sa náš algoritmus správať podobne ako Primov algoritmus. Ak vždy vyberieme najnovší prvok, bude sa správať ako rekurzívny návratový algoritmus. Taktiež môžeme vyberať najstarší pridaný prvok alebo kombinovať tieto tri spôsoby.

Ukážky bludísk vytvorených týmto algoritmom sú na obrázkoch 2.21, 2.22 a 2.23.



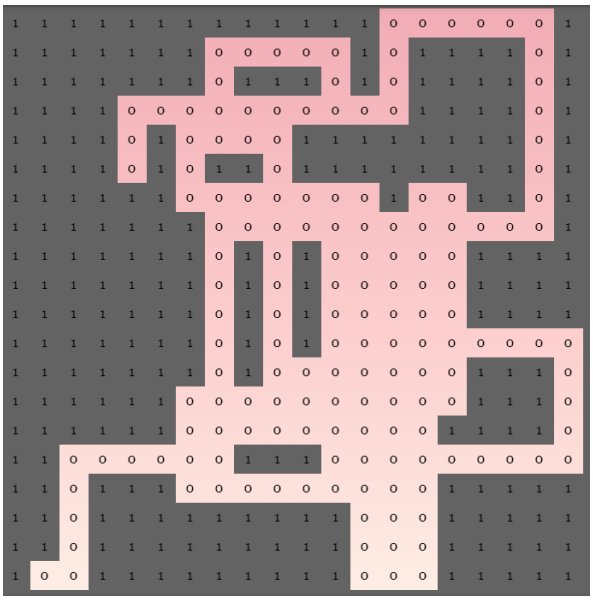
Obr. 2.23: Bludiská vytvorené rastovým algoritmom s výberom najstaršieho článku

Kapitola 3

Podzemia

Slovo „dungeon“ dlho označovalo pevnosť, hlavnú vežu hradu alebo väznice pod hradom. V dnešnej dobe však označuje aj priestory v hrách a knihách, kam sa vydávajú hrdinovia za dobrodružstvami. Pojem sa pod týmto významom ujal v sedemdesiatych rokoch minulého storočia, keď vyšla prvá edícia hry na hrdinov Dungeons & Dragons. Pôvodne šlo o jeden z možných scénárov vo vojnovej hre, kde útočníci vstúpia do hradu cez tajný vchod vo väznici a postupne sa prebojujú von. Scénár sa počas testovania stal natoľko obľúbeným, že hráči sa radšej venovali skúmaniu zákutí väznice. Tvorcovia sa toho chytili a vytvorili hru na hrdinov, ktorá sa vo forme piatej edície teší veľkej obľube aj dnes. [17]

Nie všetky typy týchto prostredí musia byť pod zemou, no kvôli jednoznačnosti budem aj tieto priestory označovať ako podzemia.



Obr. 3.1: Ukážka podzemia vytvoreného algoritmom náhodnej prechádzky

3.1 Algoritmy na generovanie podzemí

Podzemia môžu no nemusia využívať miestnosti, väčšina algoritmov, ktoré som našiel ich používa. Ak teda používame miestnosti a chodby medzi nimi, tak sú dva spôsoby akými možno generovať podzemia. Buď si najprv vytvoríme miestnosti, ktoré potom prepájame, alebo si vytvoríme chodby, ku ktorým potom vkladáme miestnosti.

3.1.1 Algoritmus náhodnej prechádzky[4]

Pri tomto algoritme postupne pridávame chodby rôznych dĺžok, ktoré idú rôznymi smermi. Budeme potrebovať rozmery cieleného podzemia, maximálny počet chodieb (tunelov) a maximálnu dĺžku každej chodby.

Začneme tým, že si vytvoríme požadované pole, ktoré si naplníme hodnotou pre stenu. Náhodne si vyberieme začiatočnú pozíciu a vytvoríme prvú chodbu náhodným smerom, náhodnej veľkosti. Posledný smer si pamätáme. Znížime maximálny počet chodieb o jedna. Kým je maximálny počet chodieb väčší ako nula, opakujeme cyklus.

Pri výbere smeru a dĺžky si dávame pozor aby sme nezobrali smer, ktorý sme použili naposledy, ani smer k nemu opačný. Vložíme chodbu do poľa. Ak sa tam nevojde celá, vložíme takú veľkú časť aká sa vojde. Ak sa nevojde vôbec, skúsime znovu vygenerovať vhodnú chodbu. Znížime maximálny počet chodieb.

Nakoniec vrátime výslednú mapu podzemia (viď. obrázok 3.1).

3.1.2 Jednoduchý smerový generátor podzemí [2]

Tento algoritmus vytvára tunel pripomínajúci jaskyňu (viď. obrázok 3.2). Vo všeobecnosti si najprv vyberieme smer, ktorým generujeme (v príklade zdola nahor) a podľa toho pridávame ďalšie priestory.

Na úvod si pripravíme tri premenné *dlzka* (aká dlhá má byť jaskyňa, t.j. počet článkov, v príklade nad sebou), *hrubost* (ako často sa mení šírka tunelu, hodnota od jedna do sto), *krivost* (ako často sa mení smer tunela v jaskyni, hodnota od jedna do sto) a pole vyplnené stenami v ktorom podzemie vygenerujeme.

Načítame si tieto hodnoty spolu so začiatočnou pozíciou x *startX*, začiatočnou pozíciou y *startY* začiatočnou šírkou *startSirka* a rozmermi podzemia. Vhodné preddefinované hodnoty sú:

- *startSirka* náhodné číslo od tri po šírku podzemia
- *startX* stred podzemia, zmenšený o polovicu premennej *startSirka*
- *startY* jeden alebo dva

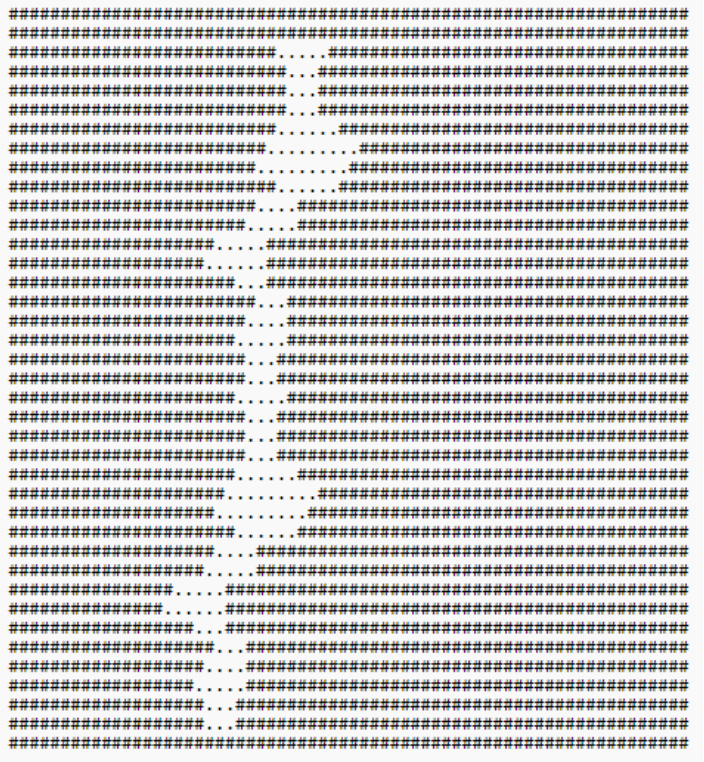
Pridáme počiatočnú miestnosť, alebo čiaru šírky *startSirka* na pozíciu *startX*, *startY*. Toto bude predstavovať vstup do jaskyne. Vytvoríme si premenné x, y a *sirka*, do ktorých v poradí vložíme hodnoty *startX*, *startY* a *startSirka*. Potom spustíme iteratívnu časť algoritmu, ktorá sa opakuje až kým nemáme dostatočnú dĺžku.

Najprv zvýšime y o jeden. Skontrolujeme, či náhodné číslo (0 až 99) je menšie ako *hrubost*. Ak áno vyberieme náhodné číslo z intervalu -2 až 2 (okrem nuly) a toto číslo pripočítame k premennej *sirka*. Pritom regulujeme premennú *sirka* aby bola minimálne tri, no maximálne rovná šírke podzemia. Potom skontrolujeme, či náhodné číslo (0 až 99) je menšie ako premenná *krivost*. Ak áno vyberieme náhodné číslo z intervalu -2 až 2 (okrem nuly) a toto číslo pripočítame k premennej x. Znovu regulujeme pozíciu aby sme nevyšli mimo hraníc podzemia. Premennú x menšiu ako nula, prepíšeme na nula, väčšiu ako šírka mapy zmenšená o tri, na šírku mapy zmenšenú o tri. Až keď máme istotu, že vložíme vhodnú čiaru prepíšeme hodnoty na súradniciach od x po x + *sirka* vo výške y na hodnotu chodby.

V tomto bode možno vložiť vylepšenia ako vkladanie tajných dverí alebo steny do stredu tunela. Nakoniec skontrolujeme, či sme už postavili dostatočne dlhú chodbu a ak áno ukončíme iteratívnu časť.

Na úplné dokončenie tohoto podzemia zvýšime y o jeden a vložíme miestnosť, alebo čiaru, ktorá predstavuje koniec jaskyne.

Jedno z možných vylepšení, ktoré môžeme jednoducho pridať je premenná *zlozitost*. Výška tejto premennej určuje koľkokrát spustíme celý algoritmus. Viacerými prechodmi vytvoríme jaskynný komplex. Nadväzujúcim rozšírením by mohlo byť spustenie algoritmu s inými smermi.



Obr. 3.2: Ukážka podzemia vytvoreného jednoduchým smerovým generátorom

3.1.3 Podzemie z miestností (Prvá možnosť) [21]

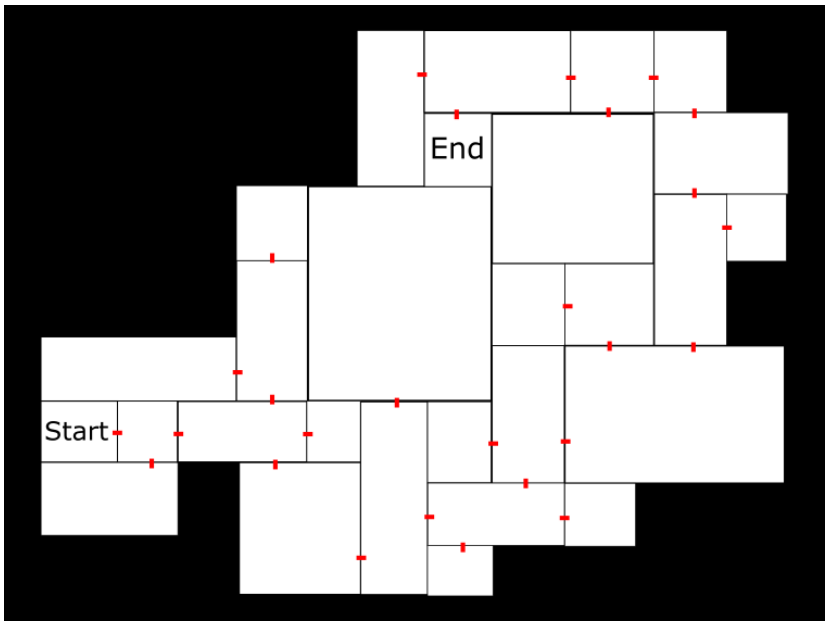
Tento algoritmus vytvára sadu miestností tesne vedľa seba. Tieto miestnosti prepája priamo dverami.

Najprv si vložíme počiatočnú a cieľovú miestnosť. Ostávajúci voľný priestor rozdelíme na obdĺžnikové miestnosti a každej z nich priradíme nejakú váhu. Potom si nájdeme najlacnejšiu cestu z počiatočnej miestnosti do cieľovej miestnosti. Miestnosti, ktorými sme prešli prepojíme dverami. Na rozšírenie podzemia, pripojíme miestnosti, ktoré sú vedľa už vytvorených miestností.

Tento algoritmus bol využitý v hre Path of Exile. Takto generované podzemia sa menia pri každej návšteve. Hra taktiež generuje niektoré vonkajšie priestory iným algoritmom. Ukážku mapy podzemia pred vyplnením článkami môžete vidieť na obrázku 3.3.

3.1.4 Podzemie z miestností (Druhá možnosť) [22]

Najprv si vytvoríme veľký počet miestností a ubezpečíme sa, že sa neprekrývajú. Potom si z týchto miestností vyberieme hlavné miestnosti. Medzi nimi si pripravíme prechody a miestnosti, ktoré nie sú hlavné a prechádzajú nimi tieto prechody zmeníme na chodby. Kde treba doplníme články tak, aby sme mali všetky miestnosti prepojené. Animáciu behu tohto algoritmu môžete vidieť na stránke zdroju[5].



Obr. 3.3: Ukážka podzemia vytvoreného z miestností prvým spôsobom

Najprv si nastavíme počet vygenerovaných miestností, napr. 150. Na tomto čísle až tak nezáleží, no vo všeobecnosti platí, že čím vyššie je toto číslo, tým väčšie a zložitejšie bude podzemie. Každú miestnosť predstavuje obdĺžnik náhodných rozmerov s obmedzenou maximálnou veľkosťou. Namiesto klasickej funkcie *random* použijeme Park-Millerove normálne rozdelenie[1]. To spôsobí, že generujeme viac malých miestností a len málo veľkých. Navyše udržujeme malý pomer rozmerov miestností. Každú z týchto miestností vložíme na náhodnú pozíciu okolo stredu podzemia.

Keď takto vytvoríme 150 miestností náhodných rozmerov, väčšina z nich sa prekrýva na malom priestore. Preto usmerníme ich oddelenie[2] tak aby sa neprekrývali. Takto ostanú vedľa seba, no pritom sa neprekrývajú. Medzery medzi miestnosťami vyplníme 1x1 veľkými miestnosťami. Týmto získame dvojrozmernú mriežku obdĺžnikov rôznych veľkostí.

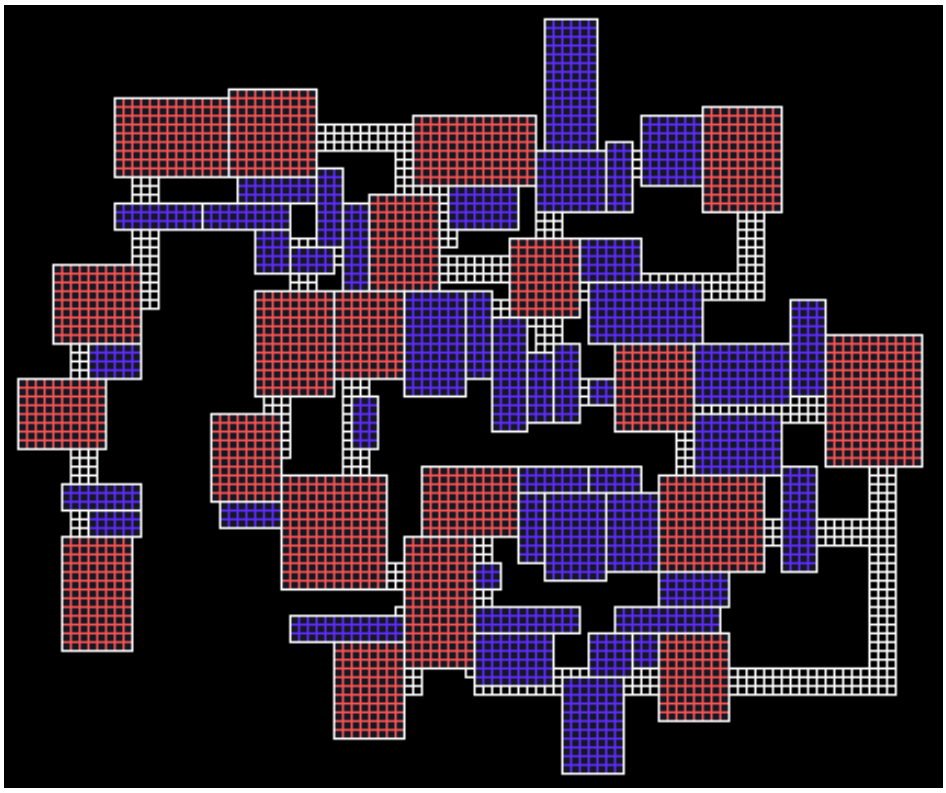
Teraz si vyberieme, ktoré z obdĺžnikov budú hlavné miestnosti. Vyberieme tie, ktorých rozmery prekračujú istú hranicu (napr. miestnosti s rozmermi väčšími ako je priemer). Týchto miestností je oveľa menej než pôvodných 150 kvôli použitiu Park-Millerovho normálneho rozdelenia. Miestnosti, ktoré sme nevybrali ešte budú mať význam, preto ich zatiaľ nemažeme. Zo stredov hlavných miestností vytvoríme graf, použitím Delaunayovej triangulácie[3]. Takto získame prepojenia medzi miestnosťami, ktoré

[1]Spôsob generovania pseudonáhodných čísel.

Každé ďalšie číslo je výsledok delenia násobku predchádzajúceho čísla konštantami, zaokrúhlené nadol na jednotky.

[2]Zistíme či sa miestnosť prekrýva s inými miestnosťami a ak áno vypočítam vektor posunu a posuniem miestnosť. Opakujeme kým sa prekrývajú miestnosti.

[3]prepojí stredy hranami tak, že žiadny z kruhov opísaných vzniknutými trojuholníkmi neprekrýva



Obr. 3.4: Ukážka podzemia vytvoreného z miestností druhým spôsobom

sa nekrižujú.

Kvôli hernej zaujímavosti vytvoreného podzemia nechceme aby bola každá miestnosť prepojená s každou susednou koridorom, preto si nájdeme minimálnu kostru tohto grafu. Takto zabezpečíme, že všetky miestnosti možno navštíviť. No takto prepojený graf je pre hru taktiež nezaujímavý, preto obnovíme cca. 8 až 15% hrán, ktoré sme odstránili. Teraz máme graf, v ktorom existuje aspoň jedna cesta z každého bodu do každého iného bodu.

Na stvárnenie grafu v podzemí využijeme koridory. Každú hranu zobrazíme ako jeden alebo dva rovné koridory[4]. Ak sa miestnosti dá spojiť len jedným koridorom, spravíme tak, ak nie vytvoríme dvojicu koridorou (tzv. L tvary), ktorými prepojíme stredy stien miestností. Teraz využijeme miestnosti, ktoré sme nevybrali za hlavné. Miestnosti, cez ktoré prechádzajú koridory sa stanú chodbami. Medzi týmito miestnosťami nepridávame dvere, jednoducho odstránime časť steny, ktorú majú miestnosti spoločnú. Rozdielnosť veľkostí obdĺžnikov spôsobí, že steny chodieb budú kľukaté a nerovnomerné. To však pridáva na zaujímavosti tohto riešenia.

Tento algoritmus vytvoril a vo svojej hre použil tvorca hry Tiny Keep, Phi Dinh. Príkladom finálnej mapy podzemia vytvoreného týmto spôsobom je obrázok 3.4.

bod mimo opísaného trojuholníka

[4]články si označíme ako časť chodieb

Obr. 3.5: Ukážka podzemia vytvoreného mriežkovým generátorom v stave pred pridaním chodieb.

3.1.5 Mriežkový generátor podzemí[3]

Pri tomto generátore si priestor najprv rozdelíme na podpriestory pevných veľkostí. Do podpriestorov vložíme miestnosti a miestnosti potom poprepájame.

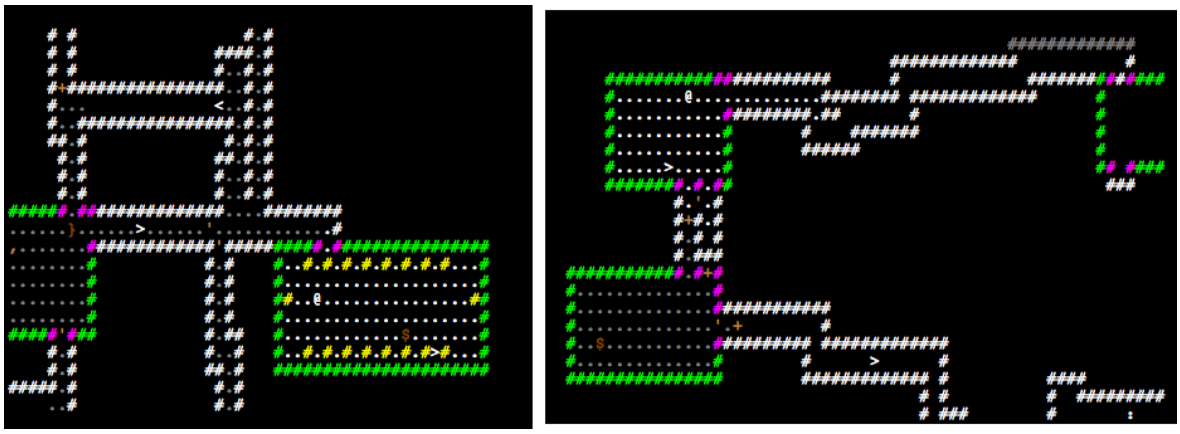
Začneme s prázdnou plochou. Túto plochu si rozdelíme na články rovnakej štvorcovej veľkosti. Keď máme pripravenú mriežku, vygenerujeme si náhodný počet miestností náhodných rozmerov, každá sa však musí zmestiť do článku. Pre každú miestnosť si náhodne vyberieme článok a umiestnime doň miestnosť, pritom označíme článok ako plný. Ak je článok už plný vyberieme si iný.

Prepojenie miestností môže prebiehať rôznymi spôsobmi popísanými v ostatných algoritmoch. V každom prípade vytvoríme chodby, pripojíme ich k miestnostiam a odstránime slepé konce (ak chceme). Mapa pred doplnením ciest môže vyzerať napr. ako obrázok 3.5.

3.1.6 Generovanie podzemí v hre Angband [16]

V hreAngband sa podzemia generujú na základe mriežkového generátoru podzemí s menšími obmenami.

Začneme s priestorom pevnej veľkosti. Tento priestor si rozdelíme na časti veľkosti



Obr. 3.6: Ukážky častí podzemia vytvoreného pre hru angband.

11x11. Vyberieme si typ miestnosti [5] a pokúsime sa takúto miestnosť vložiť do častí mriežky (miestnosť môže zaberať viac než jednu časť). Ak to nejde, pokračujeme novou miestnosťou. Ak sa to dá urobiť, postupne vykreslíme miestnosť a označíme časti ako obsadené. Každý článok nesie informáciu o type (stena, dvere, roh, a iné). Takto sa pokúsime vložiť 50 miestností. Potom okraj celého priestoru nastavíme ako nepriechodný, aby sme tam nemohli vložiť chodbu.

Keď už máme miestnosti začneme vytvárať medzi nimi chodby. Postupne sa snažíme nájsť cestu z aktuálnej miestnosti do všetkých neskoršie vytvorených miestností. Pri každej dvojici miestností sa snažíme spojiť stred aktuálnej miestnosti so stredom ďalšej miestnosti chodbou. Na aktuálnej pozícii sa vždy rozhodujeme či pokračujeme priamo smerom k miestnosti, alebo ideme náhodným smerom. Navyše si značíme keď prejdeme cez stenu, že tam treba vložiť dvere, obchádzame rohy a vyhýbame sa hranám už vložených dverí (aby neboli dvoje a viaceré dvere vedľa seba). Keď križujeme inú chodbu, zapamätáme si priesečník stien na umiestnenie dverí. Ak sa takto dostaneme do stredu hľadanej miestnosti, prekopeme chodbu, ktorú sme si značili, inak zahodíme celý postup a ideme ďalej.

Po prepojení miestností vyberieme náhodný počet objektov (príšery, pasce, trosky, a iné) a vložíme ich do miestností a chodieb. Toto vkladanie prebieha podľa hĺbky [6]. Väčšinou sú tieto objekty vložené do miestností s rovnakou hľbkou, no občas nie.

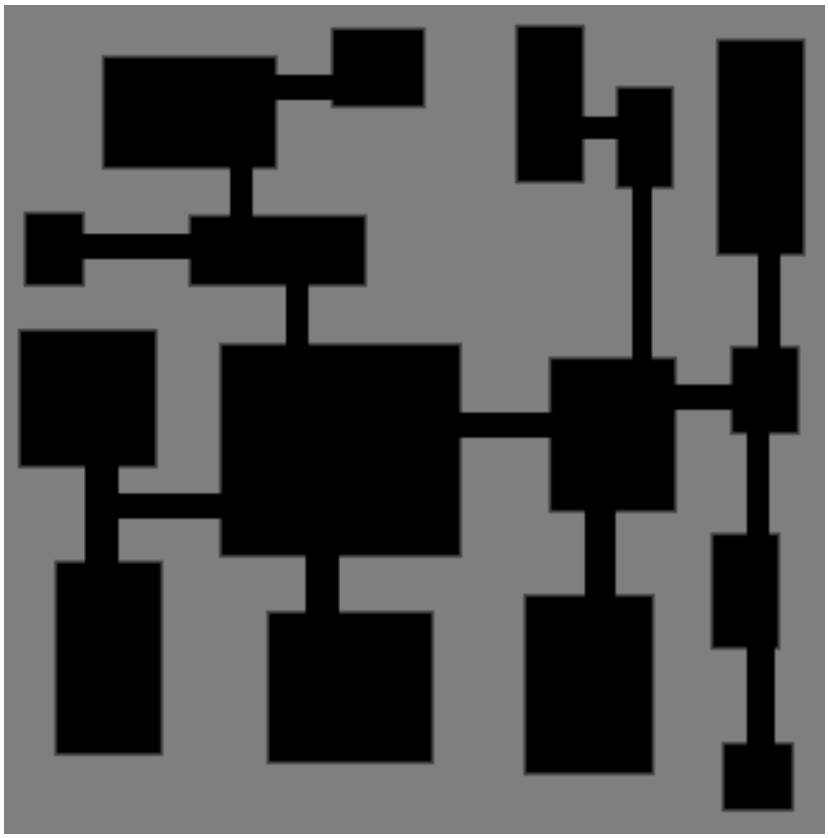
Ukážky častí podzemia vytvoreného týmto spôsobom môžete vidieť na obrázku 3.6.

3.1.7 Binárne deliaci generátor podzemí[1]

Podobne ako pri mriežkovom algoritme si pri tomto generátore najprv rozdelíme priestor na podpriestory. Tie nemusia byť rovnako veľké, ani štvorcové no musia mať vhodnú

[5] tento algoritmus potrebuje veľa predpripravených informácií, v tomto prípade viacero typov miestností, každá s inou veľkosťou a inými vlastnosťami

[6] vlastnosť miestností a objektov spájajúca vhodné dvojice



Obr. 3.7: Ukážka podzemia vytvoreného binárne deliacim generátorom

veľkosť pre miestnosti. Tentokrát využijeme všetky podpriestory a do každého vložíme miestnosť. Susedné podpriestory postupne spájame a miestnosti v nich prepájame chodbou.

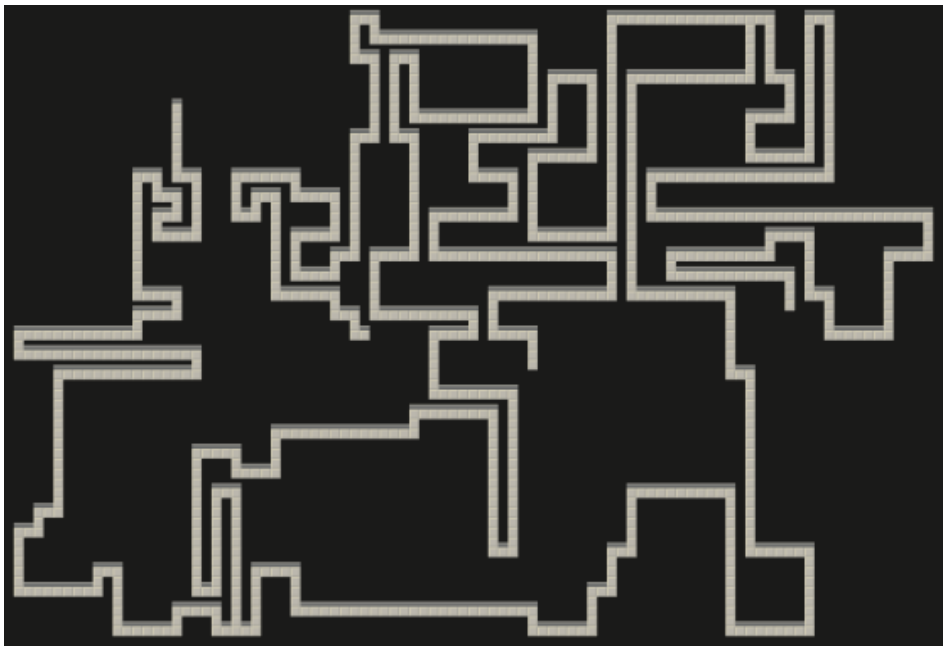
Začneme s jedným priestorom. Náhodne si vyberieme smer a pozíciu, kde a ako priestor rozdelíme. V tomto mieste rozdelíme priestor na dva podpriestory. Tento postup opakujeme s podpriestormi až kým nie sú rozmery všetkých podpriestorov vhodne veľké.

Do každého z podpriestorov vložíme miestnosť náhodných rozmerov, vrámci veľkosti podpriestoru. Zo susedných podpriestorov si spravíme dvojice. Miestnosti v každej dvojici spojíme chodbou a priestory spojíme do jedného. Opakujeme až kým znovu nemáme len jeden priestor.

Príklad mapy podzemia vytvoreného týmto algoritmom môžete vidieť na obrázku 3.7.

3.1.8 Podzemia s chodbami tvoriacimi bludisko[20]

Pri tomto type algoritmu máme na výber viacero možností. Môžeme najprv vkladať miestnosti, potom cesty, alebo s menšou úpravou naopak. Taktiež si môžeme vybrať algoritmus na generovanie bludísk, ktorým budeme vyrábať cesty.



Obr. 3.8: Ukážka podzemia vytvoreného algoritmom 3.1.8 spôsobom najprv cesty. Mapa je v stave pred vložením miestností.

Najprv cesty

Spustíme algoritmus na generovanie bludiska na celom priestore. Prekopeme ešte niektoré steny aby sme nemali perfektné bludisko. Potom odstránime slepé uličky. Vygenerujeme si miestnosti a pokúsime sa ich vhodne vložiť do priestoru[7] a prepojíme ich s už vytvoreným podzemím.

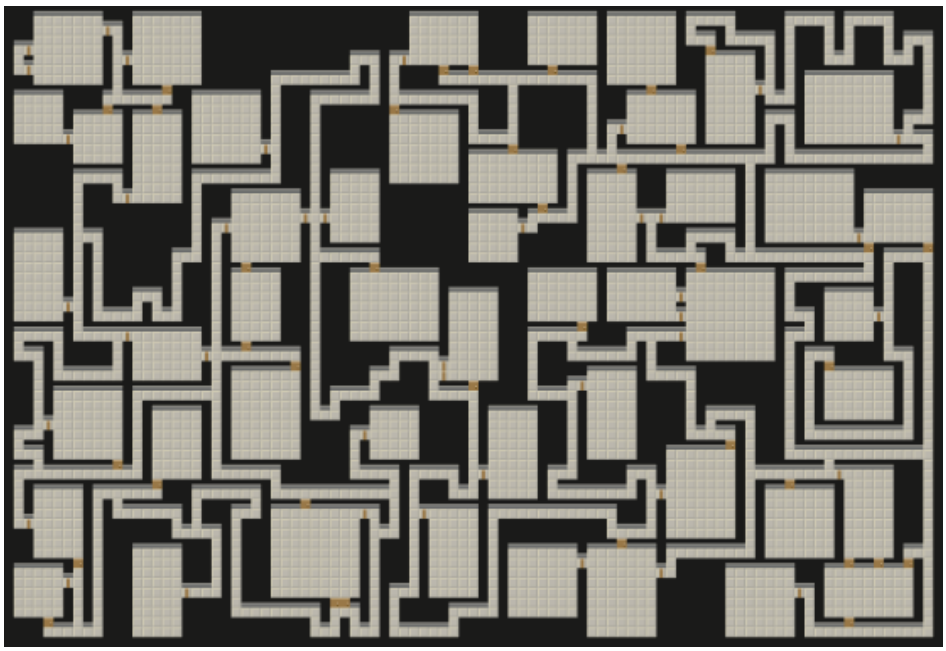
Pred vložením miestností môže mapa podzemia vyzerať napríklad ako na obrázku 3.8.

Najprv miestnosti

Vytvoríme si miestnosti náhodných rozmerov a vložíme ich do priestoru. Každú z miestností sa pokúsime vložiť do priestoru. Ak by nastal prekryv s inou miestnosťou, tento pokus zahodíme. Na voľných priestoroch spustíme algoritmus na generovanie bludiska. Nájdeme všetky miesta, ktoré sú plné a zároveň susedia s práve dvomi článkami, ktoré sú z rôznych regiónov. Vyberieme si hlavný región a jedným z jeho prepojení pripojíme ďalší región. Potom odstránime ostatné prepojenia medzi týmito regiónmi, alebo ak chceme, môžeme niektoré z nich prekopať. Tieto dva regióny sú teraz hlavný región. Opakujeme kým nemáme len jeden hlavný región. Nakoniec odstránime slepé uličky, alebo ak chceme, len časť z nich.

Po dokončení, môže mapa podzemia vyzerať ako na obrázku 3.9.

[7] Vhodne vložená miestnosť sa neprekrýva so žiadnou už vytvorenou časťou podzemia a zároveň sa jej dotýka, čiže ju môžeme jednoducho prepojiť.



Obr. 3.9: Ukážka podzemia vytvoreného algoritmom 3.1.8 spôsobom najprv miestnosti.

Kapitola 4

Implementácia

V tejto kapitole popíšem pre ktorý algoritmus som sa rozhodol a prečo, aké som v ňom uplatnil úpravy a rozpíšem vlastnú implementáciu. Moja implementácia je písaná v jazyku Python 3 v prostredí IDLE 3.6. Výstupom programu je obrázok vo formáte png, mapa vytvoreného podzemia. Pre správnu funkčnosť tvorby obrázku, musí byť na počítači nainštalovaný ImageMagick a Wand[1]. Pri inštalácii Wand stačí nasledovať pokyny k inštalácii uvedené na stránke Instalation – Wand.

[1] prepojenie ImageMagick a Pythonu

4.1 Výber

Rozhodol som sa implementovať Podzemia s chodbami tvoriacimi bludisko. Tento algoritmus som si vybral pretože chcem vytvoriť podzemie s miestnosťami a zároveň využiť algoritmy na tvorbu bludísk. Navyše ukážky k tomuto algoritmu sa najviac blížili mojej predstave mapy podzemia.

Najprv vkladám miestnosti a potom v ešte plných priestoroch spúšťam Rastový algoritmus na vytvorenie bludiska. Rastový algoritmus som si vybral, pretože je veľmi dobre prispôsobiteľný, jednoduchý na vytvorenie a pomerne rýchly.

4.2 Úpravy

Zmenil som možné typy prechodov medzi miestnosťami na obrázky dverí. Namiesto bežných chodieb (prázdny priestor) používam hodnoty od dva do trinásť, ktoré predstavujú rôzne typy dverí (dvere, zamknuté dvere, dvere s pascou, skryté dvere, padacie mreže a prechod s klenbou). Párna hodnota označuje vertikálny prechod cez dvere (zhora nadol) a hodnota o jedna väčšia predstavuje tie isté dvere pre horizontálny prechod.

Aby sme túto úpravu mohli využiť, generovanie mapy teraz používa zaužívané symboly pre pridané typy dverí.

Zistil som, že na použitie algoritmu vo forme v akej bol pôvodne by som potreboval absurdné množstvo obrázkov dverí. Preto som zmenil informáciu, ktorú nesie článok podzemia (popis v časti 4.3.4). Problém nastal pri generovaní bludiska, kde som už nemohol pridať každého suseda a nastaviť mu smery. Ak by pridanie suseda spôsobilo, že vznikne štvorica prázdnych článkov pri sebe (nezávisle od ciest medzi nimi), označím ho ako zablokovaný. To ďalej spôsobuje, že niektoré články sa stávajú nedosiahnuteľné. Takéto články odstraňujem.

Tieto úpravy spôsobili, že sa moja verzia rastového algoritmu správa inak ako originálna forma rastového algoritmu.

Meno	Hodnota	Význam
U, D, L, R	1, 2, 4, 8	hodnoty smerov hore, dole, vľavo a vpravo (Up, Down, Left, Right)
Empty, Full	0, 1	hodnoty pre prázdny a plný článok podzemia
Wall	2	pomocná hodnota, ktorá hovorí, s týmto článkom nič nerob, pred tvorbou dverí ju menením na Full
images	[img0, img1, ... img13]	pole obrázkov článkov, ktoré tvoria podzemie
imgSize	15	veľkosť strany obrázku článku v px

Tabuľka 4.1: Tabuľka konštánt použitých v implementácii a ich popis.

Premenná	Cesta k obrázku	Význam
img0	„tiles/0.png“	prázdny priestor, chodba
img1	„tiles/1.png“	plný priestor, stena
img2, img3	„tiles/2.png“, „tiles/3.png“	bežné dvere
img4, img5	„tiles/4.png“, „tiles/5.png“	zamknuté dver
img6, img7	„tiles/6.png“, „tiles/7.png“	klenutý prechod
img8, img9	„tiles/8.png“, „tiles/9.png“	dvere s pascou
img10, img11	„tiles/10.png“, „tiles/11.png“	padacia mreža
img12, img13	„tiles/12.png“, „tiles/13.png“	tajné dvere

Tabuľka 4.2: Tabuľka použitých obrázkov a ich popis. V prípade dvoch obrázkov v jednom riadku párne číslo označuje horizontálne dvere, ktorými možno prejsť zhora nadol a nepárne označuje vertikálne dvere, prechod zľava doprava. všetky dvere sú obojsmerné.

4.3 Implementácia

Popis implementácie je rozdelený na päť častí. Prvá časť popisuje importované funkcie a konštanty. Druhá časť popisuje pomocné funkcie. Tretia časť popisuje hlavné funkcie. Štvrtá časť popisuje triedu *cell*. Posledná časť popisuje beh hlavného programu, načítanie vstupu a volania funkcií.

4.3.1 Importované funkcie a konštanty

V implementácii využívam funkcie *randrange* a *shuffle* z knižnice *random* na generovanie náhodných čísel a náhodné preusporiadanie poľa. Na vytvorenie obrázku podzemia používam triedu *Image* z knižnice *wand.image*.

V programe používam konštanty popísane v tabuľke 4.1. Pole *images* obsahuje obrázky článkov. Popis týchto obrázkov je v tabuľke 4.2.

4.3.2 Pomocné funkcie

Za pomocné funkcie považujem tie, ktoré sú porovnateľne kratšie ako hlavné funkcie.

nextNum(count)

Funkcia *nextNum(count)* vracia prvok postupnosti s poradovým číslom *count*. Používa pritom pomocné pole už vygenerovaných prvkov postupnost *nextNumArr*. Ak sme toto číslo ešte nevygenerovali, spravíme tak. Hodnota požadovaného prvku je v oboch prípadoch v poli *nextNumArr* na pozícii *count*.

Nultý prvok postupnosti je nula. Hodnota i-teho prvku postupnosti je $(nextNumArr[i-1]) + (10 \cdot i)$. Túto funkciu využívame pri automatickom nastavení maximálnych rozmerov miestností.

opposite(num)

Funkcia *opposite(num)* vracia hodnotu smeru opačnú voči smeru s hodnotou *num*. V prípade, že vstupná hodnota nevyhovuje žiadnemu zo smerov, vraciame hodnotu nula.

newPos(x, y, direction)

Funkcia *newPos(x, y, direction)* nájde a vráti pozíciu prvku smerom *direction* od pozície x, y vo forme tuple. Ak vstupný smer nevyhovuje, vracia pozíciu (-1, -1), ktorá je mimo podzemia.

checkPos(x, y)

Funkcia *checkPos(x, y)* kontroluje, či je pozícia x, y medzi hranicami podzemia. Ak pozícia spĺňa tento predpoklad, vracia hodnotu **True**, inak vracia **False**.

addDoor(door)

Funkcia *addDoor(door)* vytvára na danej pozícii prechod (dvere) a upravuje hodnoty susedných článkov. Vstupná hodnota *door* je pole, ktoré obsahuje:

- door[0] = článok poľa, kam vkladáme dvere
- door[1], door[2] = indexy susedných miestností (tieto premenné v tejto funkcii nevyužívam)
- door[3] = smer od prvého suseda s hodnotou *Empty* ku dverám
- door[4] = dvojica susedov dverí s hodnotou *Empty*, door[4][0] a door[4][1] sú typu *cell*, trieda, ktorú popisujem v časti 4.3.4

- door[5] = druhý článok steny, ak je stena medzi susedmi široká dva články, inak článok prvého suseda

Najprv susedným článkom zvýšime *leaveDirs* o príslušnú hodnotu smeru. Článku dverí nastavíme *leaveDirs* na súčet oboch pridaných smerov a jeho *value* dočasne nastavíme na náhodnú percentuálnu hodnotu (0 až 99). Podľa tejto percentuálnej hodnoty vyberieme typ dverí, ktoré budú na tomto článku. Ak je táto hodnota menšia ako 75, nastavíme *value* článku dverí náhodne na hodnotu dva (bežné dvere), štyri (zamknuté dvere) alebo šesť (klenutý prechod). Ak je hodnota menšia ako 95, nastavíme *value* článku dverí náhodne na hodnotu osem (dvere s pascou) alebo desať (padacia mreža). Ak je hodnota väčšia, nastavíme *value* článku dverí na dvanásť (tajné dvere).

Typy dverí som si pomyselne rozdelil do troch kategórií podľa výskytu na bežne stretnuteľné dvere (dvere, zamknuté dvere a klenutý prechod), ťažšie stretnuteľné dvere (dvere s pascou a padacia mreža) a tajné dvere. Pravdepodobnosť výskytu bežných dverí je 75% , ťažšie stretnuteľné dvere stretneme s pravdepodobnosťou 20% a tajné dvere s pravdepodobnosťou 5%.

Keď už máme hodnotu dverí pozrieme sa znovu na smer. Ak je väčší ako tri, dvere sú vertikálne a potrebujeme zvýšiť *value* článku dverí o jedna, aby sme tu vykreslili správny obrázok.

Nakoniec skontrolujeme, či tieto dvere spájajú regióny cez stenu so šírkou dva ($door[5]! = door[4][0]$). Ak áno upravíme aj hodnoty článku *door[5]*. Jeho *leaveDirs* nastavíme na rovnakú hodnotu ako pre *door[0]* a jeho *value* nastavíme na *Empty*.

4.3.3 Hlavné funkcie

Za hlavné funkcie považujem tie funkcie, ktoré sú porovnateľne dlhšie než ostatné. Funkciu *save()* sem zaraďujem tiež, hoci je krátka pretože ju volám iba raz.

save()

Ukladanie obrázkov má na starosti funkcia *save()*. Výsledný obrázok tvorí mapa podzemia a okraj o šírke $int(imgSize/2)$px okolo nej.

Vytvoríme si obrázok *imgOut* o šírke $(Width + 1) \cdot imgSize$ a výške $(Height + 1) \cdot imgSize$. V cykle si nájdeme aktuálnu pozíciu x a y v obrázku. Ak je aspoň jedna z týchto pozícií na hranici obrázku, vkladáme na danú pozíciu v obrázku okraj, čiže *img1*. Inak vkladáme na túto pozíciu obrázok z poľa *images* podľa aktuálnej hodnoty článku podzemia. Nakoniec obrázok *imgOut* uložíme ako „mapa.png“.

addRoom(x, y, w, h)

Miestnosti do podzemia vkladáme funkciou *addRoom(x, y, w, h)*. Pred vložením miestnosti do podzemia kontrolujeme všetky články, ktoré týmto vložením prekryjeme a články okolo miestnosti[2]. Kvôli tomu si najprv nájdem hranice kontrolovaného priestoru *startX*, *startY*, *endX*, *endY* a pripravím si pole *toBlock*. Toto pole bude v prípade úspešného vloženia miestnosti obsahovať články okolo miestnosti.

V prvom cykle prejdeme kontrolované články a ak nájdeme článok, ktorého *value* je *Empty* ukončíme tento beh funkcie. Vložením miestnosti by sme totiž prekryli už vygenerovanú časť podzemia. Pokiaľ sa dostaneme za túto podmienku, a aktuálna pozícia je za hranicami miestnosti, pridám tento článok do poľa *toBlock*.

Ak sa nám podarilo ukončiť prvý cyklus a zostať vo funkcii, miestnosť môžeme vložiť. Pred vložením si pripravíme pole *cells*. Toto pole bude obsahovať články v miestnosti.

V druhom cykle prejdeme články, ktoré budú tvoriť túto miestnosť. Každý z týchto článkov si pridáme do poľa *cells*, jeho *value* nastavíme na *Empty* a jeho *leaveDirs* postupne sčítame podľa pozície v miestnosti. Po prejdení článkov miestnosti, nastavíme *value* pre všetky články v poli *toBlock* na *Wall*. Nakoniec pridáme pole *cells* (aktuálnu miestnosť) do poľa *rooms*.

grow(x, y)

Funkcia *grow(x, y)* vytvorí bludisko z ešte plných článkov, začínajúc na pozícii *x*, *y*. Táto funkcia predstavuje implementáciu rastového algoritmu s upraviteľným spôsobom voľby aktuálneho článku.

Pripravíme si pole *mazeCells*, ktoré bude obsahovať články tvoriace bludisko. Pridáme doň začiatočný článok a tomuto článku nastavíme *value* na *Empty*. Pripravíme si aj pole *cells* a pridáme mu ten istý článok. V tomto poli si udržujeme zoznam článkov bludiska, z ktorých ešte môžeme tvoriť chodby. Každý prvok tohto poľa obsahuje článok, premiešané pole indexov susedov[3] a smer, ktorým sme do článku vstúpili.

Kým pole *cells* nie je prázdne, vyberáme z neho prvok (podľa spôsobu výberu) a načítame jeho premenné. Premenná *noNewCell* slúži na kontrolu, či sa nám z aktuálneho prvku podarilo vytvoriť chodbu (pridať článok). Aktuálnemu článku skontrolujeme všetkých susedov. Ak sme sa do aktuálneho článku dostali z testovaného suseda alebo tento sused nie je *Full*, tak preskakujeme zbytok testov, odstránime index na daného suseda z poľa a opakujeme s iným susedom. Inak spočítame prázdnych susedov tohto suseda. Zapamätáme si pritom prvého prázdneho suseda rôzneho od aktuálneho článku

[2]články okolo miestnosti sú tie, ktorých vzdialenosť od miestnosti je maximálne jedna, v tomto prípade aj uhlopriečne

[3]ide o indexy vrámci poľa *c.neighbours* pre článok *c*

a smer k nemu.

Iba v dvoch prípadoch pridávam suseda do bludiska. Prvý je ak jeho počet prázdnych susedov je jeden, aktuálny článok. Druhý prípad nastáva ak má sused práve dvoch prázdnych susedov a rozhodneme sa spraviť cyklus. Cyklus spravíme ak vygenerované náhodné číslo od 0 do 99 je menšie ako percentuálna šanca *Percent* nastavená podľa vstupu od používateľa.

Ak sme vytvorili cyklus upravíme *leaveDirs* susedného článku a susedovho suseda aby odpovedali prepojeniu medzi nimi. Následne v oboch prípadoch nastavíme *value* susedného článku na *Empty* a *leaveDirs* susedného aj aktuálneho článku zvýšime o smer od jedného k druhému. Keďže sme pridali nový článok, zmeníme hodnotu *noNewCell* na **False**. Nakoniec pridáme článok suseda, indexy na jeho susedov a smer, ktorým sme doň vošli do poľa *cells* a aktuálnemu článku odstránime index na tohto suseda.

Po skontrolovaní všetkých susedov sa pozrieme na premennú *noNewCells* a na ostávajúci počet indexov na susedov aktuálneho článku. Ak sme nepridali nový článok alebo už neostávajú žiadne susedné články, odstránime aktuálny článok z poľa *cells*. Po vyprázdnení poľa *cells* pridáme pole *mazeCells* ako miestnosť do poľa *rooms*.

4.3.4 Trieda cell

Na uloženie informácií o podzemí som vytvoril triedu *cell*. Premenná tohto typu nesie informácie o svojej pozícii v podzemí *self.position*, svoju hodnotu *self.value*, súčet smerov, ktorými možno opustiť tento článok *self.leaveDirs*, pole susedných článkov *self.neighbours* a pole smerov k týmto susedom *self.neighbourDirs*. Pri inicializácii premennej tohto typu potrebujem len pozíciu. Jedinou ďalšou funkciou je *addNeighbour()*, ktorá článku naplní polia *self.neighbours* a *self.neighbourDirs*.

4.3.5 Hlavný program

V hlavnom programe sa postupne pýtame otázky na načítanie vstupu. Pre rýchlejšie testovanie som v programe nechal odkomentovaný alternatívny spôsob vstupu, kde sa dá vstupné údaje napísať priamo v programe.

Po načítaní vstupu sa snažíme vložiť predurčený počet miestností náhodných rozmerov do podzemia. Potom si v jednom cykle nájdeme pozície, ktoré zostali plné. V druhom cykle sa na tieto pozície pozeráme znovu a ak je aktuálna pozícia ešte plná spúšťame na nej algoritmus na generovanie bludísk.

Po vložení posledného bludiska, hľadáme vhodné miesta kde by sme bludiská a miestnosti prepojili dverami. Náhodne vyberieme jednu z miestností a označíme ju ako prepojenú. Kým sú nejaké dvere, z ktorých môžeme vyberať alebo ešte nie sú prepojené všetky miestnosti so všetkými bludiskami, hľadáme medzi dverami také, ktoré spájajú už prepojenú časť s neprepojenou časťou. Po pripojení novej časti zahadzujeme ostatné

prepojenia medzi novopripojenou časťou a už prepojeným podzemím. Pred samotným odstránim hodnoty z poľa, však dvere s pravdepodobnosťou 10% vytvoríme.

Po prepojení celého podzemia, odstránime slepé chodby podľa hodnoty premennej *deadEnds* a výsledok uložíme.

Popri texte uvádzam príklad behu môjho programu v obrázkoch vždy po zmene väčšej časti podzemia. Rozmery podzemia sú 15x15 článkov. Maximálna veľkosť miestností je automaticky nastavená na 7. Počet pokusov je 34 (z rozsahu 3 až 50). Percentuálna šanca vytvoriť cyklus je 30%. Vo funkcii *grow* najprv vyberám najstarší prvok a na konci odstránim všetky slepé chodby.

Načítanie vstupu

Prvá otázka zisťuje *Width* a *Height*, rozmery podzemia v počte článkov. Očakávame dve čísla oddelené medzerou. Limit na minimálny rozmer je nastavený na dva články. Ak by odhadovaný čas generovania podzemia daných rozmerov prekročil hodinu, pýtame si iné rozmery. Pre rozmery, ktorých odhadovaná dĺžka generovania je kratšia, no sú aj tak pomalé, vypíšeme upozornenie a spýtame sa, či chce používateľ aj tak pokračovať.

Používateľ si navyše môže vybrať aby sme mu rozmery podzemia náhodne vybrali my. V tomto prípade náhodne vygenerujeme výšku a šírku v rozsahu 10 až 200 tak aby bol ich pomer aspoň dve ku piatim. Po vygenerovaní tieto rozmery vypíšeme.

Druhou otázkou sa pýtame na dvojicu *maxWidth*, *maxHeight*, horný limit veľkostí miestností v počte článkov. Očakávame dve čísla oddelené medzerou. Tieto rozmery sú limitované zdola premennými *minWidth* a *minHeight* a zhora už pripravenými premennými *Width* a *Height*. Premenné *minWidth* a *minHeight* sú nastavené na tri. V prípade malých rozmerov podzemia (2x2) sú *minWidth* a *minHeight* nastavené na jedna.

Ak je dolný limit prekročený, varujeme užívateľa pred následkami. Ak je maximálna veľkosť miestnosti menšia ako jej minimálna veľkosť, žiadnu miestnosť nevygenerujeme. Spýtame sa používateľa, či chce aj tak pokračovať. Ak nie otázku opakujeme, inak pokračujeme ďalej. Ak je prekročený horný limit, upravíme premenné na maximálnu možnú hodnotu.

Používateľ si navyše môže vybrať automatické nastavenie týchto premenných. V tomto prípade si pre šírku aj výšku nastavíme *factor* a *num* na nula. Kým platí $Width > num$ (resp. $Height > num$) zvýšime *factor* o jedna a voláme funkciu $num = nextNum(factor)$. Nakoniec nastavíme $maxWidth = int(Width/factor)$ (resp. $maxHeight = int(Height/factor)$) a tieto hodnoty vypíšeme.

Tretím vstupom načítame premennú *tries*. Jej hodnota predstavuje počet pokusov o vloženie náhodnej miestnosti. Pokiaľ používateľ nastavil *maxWidth* alebo *maxHeight* na príliš malé, netvoríme žiadne miestnosti a tento vstup preskakujeme. V opačnom

prípade sa spýtame ako husté má podzemie byť.

Používateľ má na výber z piatich možností. Pre každú z týchto možností mierne inak vypočítame minimálny *minTries* a maximálny *maxTries* počet pokusov na vloženie miestností. Pokiaľ užívateľ zadal niečo čo nie je medzi možnosťami, znovu sa spýtame otázku.

Po výbere jednej z možností náhodne vygenerujeme hodnotu premennej *tries* v rozsahu *minTries*, *maxTries*.

Štvrtou otázkou zistíme *cellPicker*, spôsob akým budeme vyberať aktuálny článok vo funkcii *grow*. Tu používateľ dostáva na výber z piatich možností. Ku každej je krátky popis čo robí a aký to má dopad na konečný výzor podzemia.

Piatou otázkou zistíme *Percent*, percentuálna pravdepodobnosť vytvorenia cyklu vo funkcii *grow*. Používateľ má na výber zo šiestich možností: žiadne cykly (0%), málo cyklov (10%), zopár cyklov (20%), veľa cyklov (30%), veľmi veľa cyklov (50%) alebo všetky, čo sa dá (100%).

Šiestym a posledným vstupom od používateľa určíme premennú *deadEnds*. Táto premenná určuje počet slepých chodieb, ktoré po vytvorení podzemia odstránime. Používateľ si môže vybrať či nechce odstrániť žiadne, chce odstrániť zopár, veľa alebo všetky slepé chodby.

Pokiaľ sled odpovedí od používateľa spôsobí, že odstránime všetko, čo sme vytvorili, varujeme používateľa a spýtame sa ho či chce aj tak pokračovať. Ak nie, pýtame sa znovu, inak pokračujeme v programe. Ak je jedna z maximálnych veľkostí miestností primalá a *Percent* je nula, tak vytvoríme perfektné bludisko (viď. section 2.1).

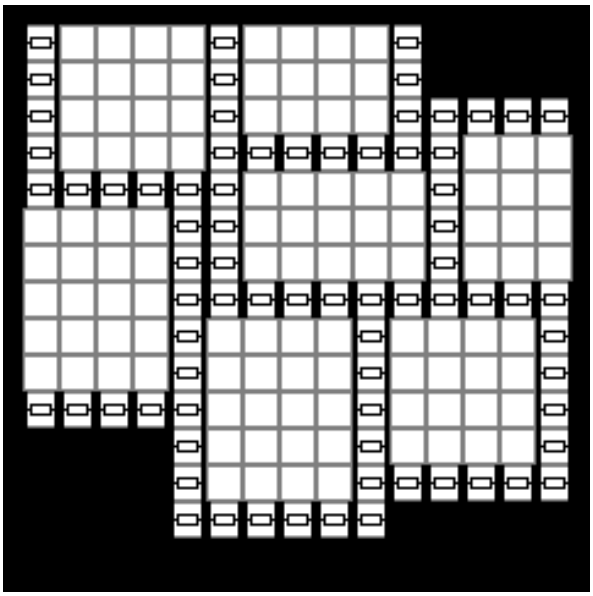
Generovanie miestností a chodieb

Po načítaní vstupov si pripravíme dvojrozmerné pole *dungeon[Height][Width]* a naplníme ho prvkami typu *cell*. Potom všetkým týmto prvkom nastavíme hodnoty v poliach cez funkciu *cell.addNeighbours()*. Pripravíme si prázdne pole *rooms* a ak sú vstupy vhodné začneme vkladať miestnosti. Pre každý z pokusov v rozsahu *tries* si najprv pripravíme šírku ($curW$) a výšku($curH$) v nastavenom rozsahu a potom nájdeme jednu z pozícií (x, y), kam sa miestnosť týchto rozmerov vojde. Aktuálny pokus vložiť miestnosť spustíme zavolaním funkcie *addRoom(x, y, curW, curH)*.

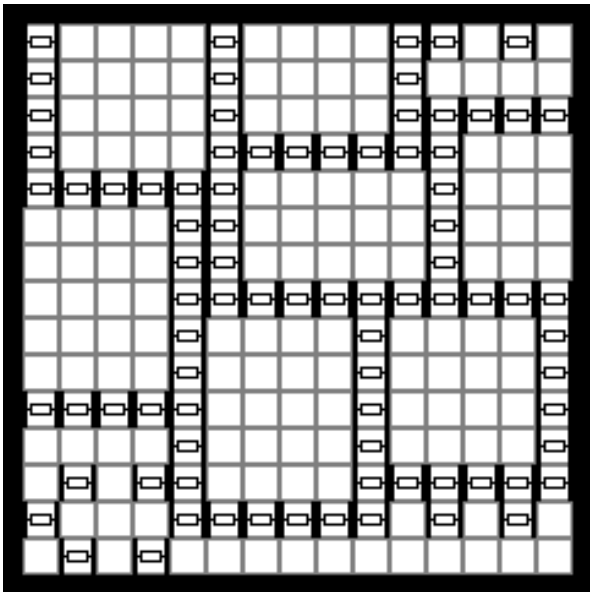
Po vložení miestností vyzerá náš príklad ako na obrázku 4.1.

V premennej *roomCount* si zapamätáme počet vložených miestností. Následne prejdeme celé pole a pozície, kde je *value* článku podzemia *Full* si uložíme do poľa *toMaze*. Premiešame toto *toMaze* a kým obsahuje prvky, postupne vyberáme pozície. Ak je článok na tejto pozícii stále *Full*, spustíme na ňom funkciu *grow* a potom pozíciu z poľa *toMaze* odstránime. Inak pozíciu len odstránim.

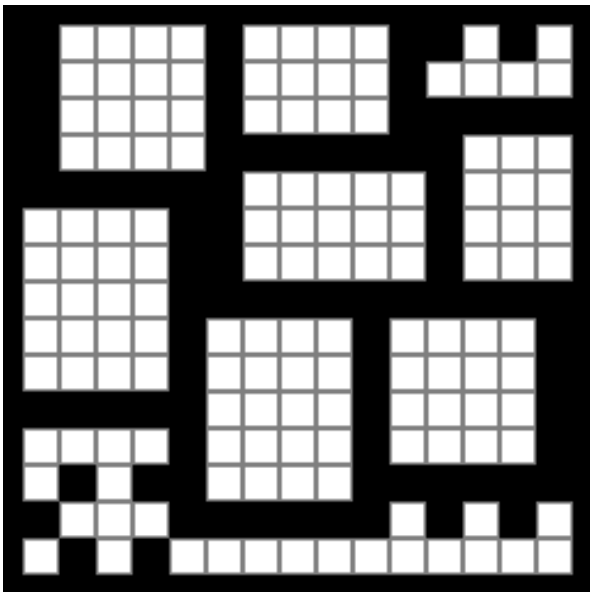
Vrámci funkcií *addRoom* a *grow* sme naplnili pole *rooms*. Od tohto momentu pova-



Obr. 4.1: Po vložení miestností obsahuje podzemie dané miestnosti a články okolo miestností. Články okolo miestností sú nastavené na hodnotu 2, čo pri pokuse o vykreslenie obrázku na danom mieste vykreslí štandardné dvere v horizontálnej polohe.



Obr. 4.2: Po vložení chodieb obsahuje podzemie dané miestnosti, chodby a články okolo nich. Články okolo nich sú nastavené na hodnotu 2, čo pri pokuse o vykreslenie obrázku na danom mieste vykreslí štandardné dvere v horizontálnej polohe.



Obr. 4.3: Po úprave hodnôt už podzemie nevykresľuje všade dvere.

žujeme všetky zoznamy článokv v tomto poli za miestnosti. Náš príklad sa vytvorením chodieb zmenil na obrázok 4.2.

Hľadanie dverí

V tomto bode sú všetky hodnoty článkov podzemia buď *Empty* alebo *Wall*. Keďže hodnota *Wall* neskôr predstavuje hodnotu dverí a svoj účel už splnila, prepíšeme všetky jej výskyty na hodnotu *Full* (po úprave sa náš príklad zmenil na obrázok 4.3).

Nájdeme vhodné pozície pre dvere a zapamätáme si ich v poli *doors*. Vhodné pozície hľadáme v cykle a pozeráme sa len na články, ktorých aktuálna hodnota je *Full*. Počas aktuálneho prechodu cyklu si článok dverí pamätáme v premennej w, a jeho pozícia v podzemí je $[j, i]$. Vytvoríme si zoznam prázdnych susedov tohto článku *neighs* a zapamätáme si smer od prvého z týchto susedov k aktuálnemu článku ako *direction*. Nájdeme pozíciu *nxt* zavolaním funkcie $nxt = newPos(j, i, direction)$. Na tejto pozícii by mal byť druhý prázdny sused článku w. Ak zistíme, že pozícia *nxt* je za hranicami podzemia, aktuálny článok nemá prázdnych susedov alebo ich má viac ako dvoch začneme od začiatku s ďalším článkom. Inak nájdeme pre prvého prázdneho suseda index miestnosti, v ktorej sa nachádza.

Nastavíme si premennú *extra* na článok prvého prázdneho suseda *neighs[0]*. Teraz môžu nastať dva prípady. V jednoduchšom prípade má w dvoch prázdnych susedov. Stačí keď skontrolujeme, či pozícia druhého suseda je práve pozíciou *nxt* a nájdeme index miestnosti druhého suseda.

Trochu zložitejší je prípad, keď má w jedného prázdneho suseda. Stena medzi miestnosťami môže byť hrubá dva články, preto skontrolujeme či nastal tento prípad. Ak je

článok na pozícii *nxt* musí byť plný. Nájdeme si pozíciu článku za ním, *nxt2* a ak je táto pozícia v rozsahu podzemia a je prázdna, tak je stena dvojitá.

Ak sme zistili, že stena je dvojitá, pridáme článok na pozícii *nxt2* do poľa prázdnych susedov (*neighs*), nájdeme si index druhej miestnosti a *extra* zmeníme na článok na pozícii *nxt* (tento článok je druhým článkom steny medzi prázdnymi susedmi).

Keď sme skontrolovali obe možnosti, pridáme pole, ktoré obsahuje článok *w*, indexy miestností, smer *direction*, pole prázdnych susedov *neighs* a článok *extra* do poľa *doors*. Pole *doors* obsahuje prvky vo formáte popísanom v časti 4.3.2.

Pridávanie dverí

Pred prvým výberom si pole *doors* premiešame a vytvoríme pole *connected*, v ktorom si budeme pamätať indexy už prepojených miestností. Náhodne si vyberieme jednu z obdĺžnikových miestností a označíme ju ako prepojenú (ak sme nevytvorili žiadne takéto miestnosti, vyberieme prvú miestnosť).

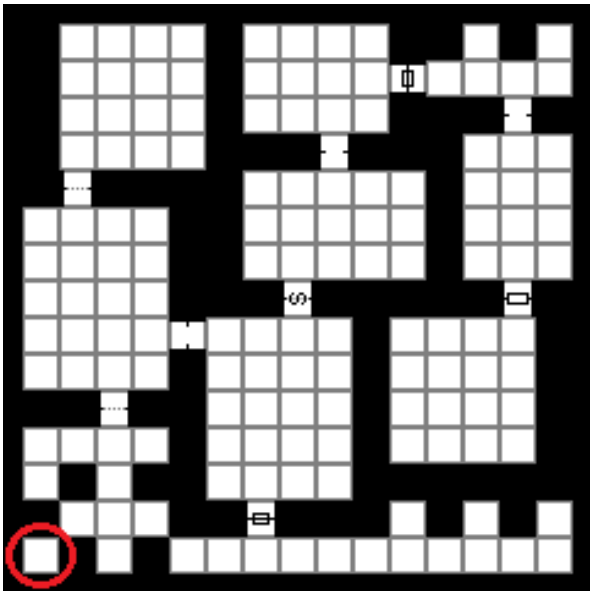
Kým nemáme prepojené všetky miestnosti, alebo neminieme všetky dvere, beží cyklus na výber a tvorbu dverí. Nájdeme si prvé dvere, ktorých jedna z miestností je už prepojená a zapamätáme si index druhej miestnosti v premennej *r2*. Nájdené dvere pridáme funkciou *addDoor*.

Po pridaní dverí odstránime ostatné dvere, ktoré by pripájali miestnosť *r2*. Prejdeme všetky indexy prepojených miestností a hľadáme dvere, ktoré spájajú tieto miestnosti s miestnosťou *r2*. Práve pridané dvere len odstránim z poľa *doors*, pri ostatných je šanca, že ich najprv vytvorím. Najprv skontrolujeme, či je stále vhodné vložiť tieto dvere. Spočítame počet prázdnych susedov dverí. Ak sú dvere dvojité a majú jedného prázdneho suseda alebo ak dvere nie sú dvojité a majú dvoch prázdnych susedov, tak vieme, že ich susedné dvere sme ešte nevytvorili a môžeme vytvoriť tieto dvere. Keďže nechceme pridať každé vhodné dvere, znižujem pravdepodobnosť na vytvorenie dverí na 7%. Ak sa rozhodneme dvere pridať, zavoláme funkciu *addDoor* na tieto dvere. Následne dvere odstránim z poľa *doors*.

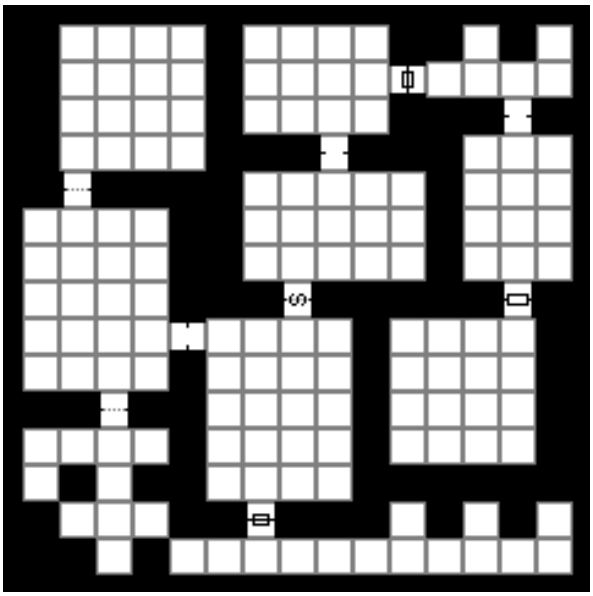
Nakoniec pridáme index miestnosti *r2* do poľa *connected*. Po pridaní dverí sa náš príklad zmenil na obrázok 4.4 zvýraznený článok je nedosiahnuteľný, v ďalšom kroku ho odstránime.

Odstránenie slepých chodieb

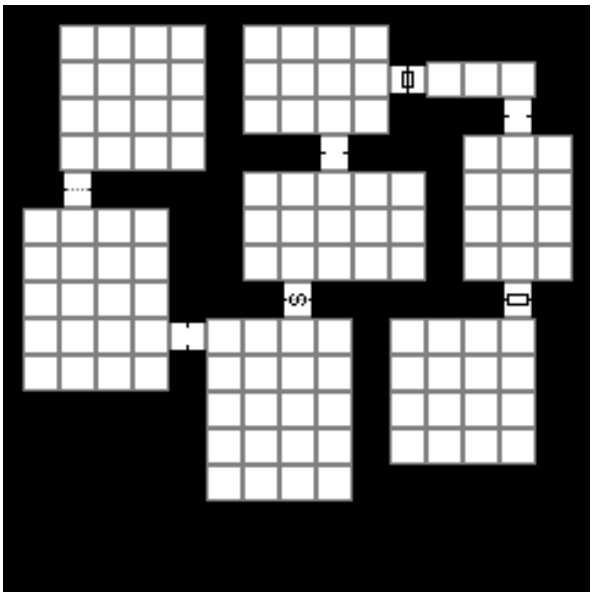
Ešte pred odstraňovaním slepých chodieb sa pozrieme na miestnosti. Ak nájdeme miestnosť, ktorú sa nedalo pripojiť, zmeníme *value* jej článkov na *Full* a ich *leaveDirs* späť na nulu. Takéto miestnosti mohli vzniknúť pri tvorbe chodieb. Ide o chodby, ku ktorým by dvere susedili s viac ako dvomi prázdnymi susedmi. Ak by sme tento krok nespravili, nie je zaručené, že ich odstránime. Náš príklad mal tiež jednu nedosiahnuteľnú časť,



Obr. 4.4: Po pridaní dverí obsahuje podzemie dané miestnosti, chodby a dvere medzi nimi. V podzemí mohli vzniknúť nedosiahnuteľné úseky, jeden taký je zvýraznený na obrázku.



Obr. 4.5: Po odstránení nedosiahnuteľných častí môžeme odstraňovať slepé konce.



Obr. 4.6: Po odstránení slepých koncov, máme podzemie hotové.

viď. obrázok 4.5.

Počet už odstránených slepých koncov si pamätáme v premennej *counter*. Aby cyklus správne bežal, si definujeme pole *neighbours* ešte pred spustením cyklu. V tomto poli udržujeme články susediace so slepými koncami a smery od nich k danému slepému koncu.

Kým neodstránime dostatok slepých koncov alebo všetky, beží cyklus **while**. Najprv si nájdeme články, ktoré treba odstrániť. Pri prvom prechode prechádzame celým podzemím. Ak je hodnota *leaveDirs* aktuálneho článku jedna z hodnôt smeru, je článok slepým koncom. Zapamätáme si článok v poli *toRem* a dvojicu tvorenú jeho susedom daným smerom a opačný smer[4] si uložíme do poľa *neighbours*.

Pri ďalších prechodoch takto kontrolujeme len články z poľa *neighbours*. Pritom nových susedov si ukladám do pomocného poľa a po nájdení slepých koncov tieto hodnoty presuniem do poľa *neighbours*.

Ak je pole *toRem* prázdne, odstránili sme už všetky slepé konce a ukončíme cyklus. Inak si prejdeme nájdené slepé konce a pre každý z nich, upravíme *value* tohto článku na *Full* a *leaveDirs* na nulu. Článku zapamätaného suseda znížime *leaveDirs* o zapamätanú hodnotu smeru. Ak predstavoval článok tohto suseda dvere, tak zmeníme jeho *value* na *Empty*. Zvýšime hodnotu *counter* o jedna a ak sme dosiahli dostatok odstránených slepých koncov ukončíme cyklus. Inak opakujeme s ďalším slepým koncom.

Nakoniec zavoláme funkciu *save()*, ktorá uloží mapu výsledného podzemia. Náš príklad dosiahol konečnú podobu v obrázku 4.6.

[4]pamätáme si opačný smer, pretože ten predstavuje hodnotu, o ktorú potrebujeme znížiť *leaveDirs* článku suseda

Záver

V prvej kapitole som popísal, čo sú bludiská, kde ich môžeme nájsť a akými algoritmami ich môžeme generovať. Bludiská môžeme rozdeliť podľa ich vlastností na rôzne kategórie. S ohľadom na generovanie bludísk patria medzi najdôležitejšie vlastnosti dimenzia, trasa, zameranie a jednotnosť (podvlastnosť štruktúry).

Algoritmy na generovanie bludísk sú: rekurzívny návratový algoritmus, Ellerov algoritmus, Kruskalov randomizovaný algoritmus, Primov algoritmus, rekurzívne delenie, Aldous–Broderov algoritmus, Wilsonov algoritmus, vyhľadávací algoritmus a rastový algoritmus.

Ako sme videli pri Aldous–Broderovom a Wilsonovom algoritme, vytvárať jednotné bludiská nie je jednoduché a nie každý algoritmus to zvláda.

V druhej kapitole som vysvetlil pôvod zmyslu slova podzemie (vrámci tejto práce) využívaného v hrách a knihách a popísal som algoritmy na generovanie podzemí.

Algoritmy na generovanie podzemí sú: algoritmus náhodnej prechádzky, jednoduchý smerový algoritmus, podzemie vytvorené z miestností (dva spôsoby), mriežkový generátor podzemí, generátor podzemí v hre Angband, binárne deliaci generátor podzemí a podzemia s chodbami tvoriacimi bludisko.

Väčšina z nich stavia podzemia s použitím alebo priamo z miestností, no podzemia bez miestností sú tiež zaujímavé.

V tretej kapitole som sa venoval mojej implementácii algoritmu „podzemia s chodbami tvoriacimi bludisko“ s využitím rastového algoritmu na tvorbu chodieb. Po zmene spôsobu ukladania informácie o článku som ešte nevedel koľko to spôsobí problémov. Nakoniec som to musel vyriešiť odstránením už vytvorených častí podzemia.

Príliš neskoro som našiel jeden krajný prípad, ktorý moja implementácia ignoruje. Presnejšie vo funkcii *grow* môže nastať prípad, kedy by sme mohli vytvoriť cyklus, no nespravíme tak. Program aj bez tohto krajného prípadu beží bez problémov, no tento prípad plánujem v dohľadnej dobe ošetriť.

Keďže narozdiel od popísaných algoritmov využívam priamo články ako steny, správa sa rastový algoritmus rozdielne. Preto plánujem zistiť ako sa program správa s inými algoritmami na generovanie bludísk a podzemí.

Tento program plánujem s menšími úpravami použiť v prípade, že by som vo svojom voľnom čase začal pracovať na vlastnej hre. Okrem toho ho budem využívať pri stolnej

hre na hrdinov, ktorú spolu s priateľmi hráme.

Literatúra

[1] Basic bsp dungeon generation. http://www.roguebasin.com/index.php?title=Basic_BSP_Dungeon_generation. [Online; accessed 22-April-2019].

[2] Basic directional dungeon generation. http://www.roguebasin.com/index.php?title=Basic_directional_dungeon_generation. [Online; accessed 13-April-2019].

[3] Grid based dungeon generator. http://www.roguebasin.com/index.php?title=Grid_Based_Dungeon_Generator. [Online; accessed 21-April-2019].

[4] Ahmad Abdolsaheb. How to code your own procedural dungeon map generator using the random walk algorithm. https://medium.freecodecamp.org/how-to-make-your-own-procedural-dungeon-map-generator-using-the-random-walk- 2017. [Online; accessed 12-April-2019].

[5] A. Adonaac. Procedural dungeon generation algorithm. https://www.gamasutra.com/blogs/AAdonaac/20150903/252889/Procedural_Dungeon_Generation_Algorithm.php, 2015. [Online; accessed 15-April-2019].

[6] Jamis Buck. Maze generation: Eller's algorithm. http://weblog.jamisbuck.org/2010/12/29/maze-generation-eller-s-algorithm, 2010. [Online; accessed 20-March-2019].

[7] Jamis Buck. Maze generation: Recursive backtracking. http://weblog.jamisbuck.org/2010/12/27/maze-generation-recursive-backtracking, 2010. [Online; accessed 20-March-2019].

[8] Jamis Buck. Maze algorithms. https://www.jamisbuck.org/mazes/, 2010 - 2011. [Online; accessed 20-March-2019].

[9] Jamis Buck. Maze generation: Aldous-broder algorithm. http://weblog.jamisbuck.org/2011/1/17/maze-generation-aldous-broder-algorithm, 2011. [Online; accessed 20-March-2019].

[10] Jamis Buck. Maze generation: Growing tree algorithm. http://weblog.jamisbuck.org/2011/1/27/maze-generation-growing-tree-algorithm, 2011. [Online; accessed 20-March-2019].

[11] Jamis Buck. Maze generation: Hunt-and-kill algorithm. http://weblog.jamisbuck.org/2011/1/24/maze-generation-hunt-and-kill-algorithm, 2011. [Online; accessed 20-March-2019].

[12] Jamis Buck. Maze generation: Kruskal's algorithm. http://weblog.jamisbuck.org/2011/1/3/maze-generation-kruskal-s-algorithm, 2011. [Online; accessed 20-March-2019].

[13] Jamis Buck. Maze generation: Prim's algorithm. http://weblog.jamisbuck.org/2011/1/10/maze-generation-prim-s-algorithm, 2011. [Online; accessed 20-March-2019].

[14] Jamis Buck. Maze generation: Recursive division. http://weblog.jamisbuck.org/2011/1/12/maze-generation-recursive-division-algorithm, 2011. [Online; accessed 20-March-2019].

[15] Jamis Buck. Maze generation: Wilson's algorithm. http://weblog.jamisbuck.org/2011/1/20/maze-generation-wilson-s-algorithm, 2011. [Online; accessed 20-March-2019].

[16] Andrew Doull. Unangband dungeon generation - parts 1-9. http://roguelikedeveloper.blogspot.com/2007/11/unangband-dungeon-generation-part-one.html, 2007 - 2008. [Online; accessed 20-April-2019].

[17] KorvinStarmast. Why does 'dungeon' mean the places adventurers go to kill stuff? https://rpg.stackexchange.com/questions/67422/why-does-dungeon-mean-the-places-adventurers-go-to-kill-stuff, 2015. [Online; accessed 10-April-2019].

[18] Hong Minhee. Instalation – wand 0.4.1. http://docs.wand-py.org/en/0.4.1/guide/install.html, 2015. [Online; accessed 17-May-2019].

[19] Josh Nilaya. The a-maze-ing history of mazes and labyrinths. http://www.wnpr.org/post/maze-ing-history-mazes-and-labyrinths, 2018. [Online; accessed 2-March-2019].

[20] Bob Nystrom. Rooms and mazes: A procedural dungeon generator. http://journal.stuffwithstuff.com/2014/12/21/rooms-and-mazes/, 2014. [Online; accessed 2-May-2019].

[21] Jonathan Rogers (Path of Exile). Path of exile - random level generation presentation. https://www.youtube.com/watch?v=GcM9Ynfzll0, 2011. [Online; accessed 14-April-2019].

[22] Phi Dinh (@phi6). Procedural dungeon generation algorithm explained. https://www.reddit.com/r/gamedev/comments/1dlwc4/procedural_dungeon_generation_algorithm_explained/, 2013. [Online; accessed 15-April-2019].

[23] Walter D. Pullen. Maze classification. http://www.astrolog.org/labyrnth/algrithm.htm, 2015. [Online; accessed 3-March-2019].