Generowanie nieskończonych poziomów jaskiń przy użyciu Automatów Komórkowych

(Generating Infinite Cave Levels using Cellular Automata)

Cezary Kosiński

Praca licencjacka

Promotor: dr Jakub Kowalski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

30.08.2019

Streszczenie

W mojej pracy implementuję generator map do gier typu roguelike opierający się na automacie komórkowym. W celu znalezienia optymalnych parametrów automatu komórkowego został zastosowany algorytm ewolucyjny ewaluujący jakość generowanych map. Generator ma teoretyczną możliwość tworzenia nieskończonych, złożonych labiryntów. Zaprezentowane wyniki pokazują, że metoda ta, po uprzednim udoskonaleniu procesu ewolucji, przynosi zamierzone efekty.

The goal of my thesis is to implement a cellular automaton based rogue-like map generator. In order to optimise the results, evolutionary algorithm was used to evolve parameters of cellular automaton. In theory, the generator is able to produce infinite, complex labirynths. Presented results shows, that after additional improvements in the process of evolution, presented method meets the assumptions.

Spis treści

Rozdział 1.

Wstęp

Bezpośrednią inspiracją do napisania niniejszej pracy była publikacja Johnsona i wsp. [12] traktująca o generowaniu automatem komórkowym nieskończonych poziomów jaskiń w czasie rzeczywistym. Plansze takie mogą być zastosowane w różnego rodzaju grach o charakterze eksploracyjnym, oferując graczowi potencjalnie nieograniczoną zawartość.

W rozdziale 2. omówione są: geneza pojęcia proceduralnego generowania treści wraz z jego historią; przykładowe zastosowania tej metody w grach komputerowych oraz propozycje ich klasyfikacji; jej potencjalne wady na podstawie przykładów z branży gier oraz jej rolę w budowaniu labiryntów. Rozdział 3. wprowadza pojęcia takie jak automaty komórkowe i algorytmy ewolucyjne, opisując tym samym ogólne idee metod, do których bezpośrednio odnosi się dalsza część pracy. Ostatni (4.) z rozdziałów stanowiących trzon pracy prezentuje implementację autorskiego programu wraz z deskrypcją eksperymentów, wyników w nich uzyskanych oraz płynących z nich wniosków.

Rozdział 2.

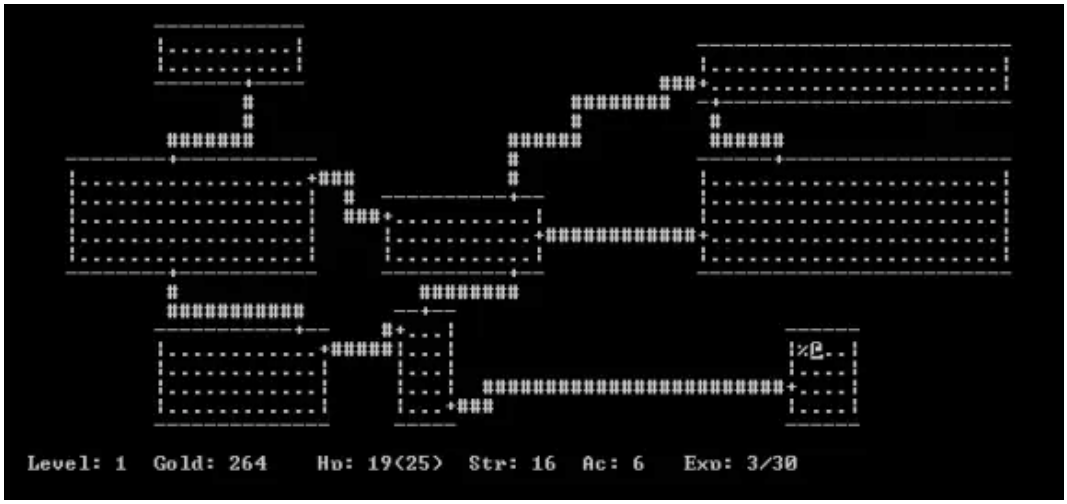
Proceduralne generowanie zawartości

Proceduralne generowanie zawartości (ang. PCG – procedural content generation) jest dziedziną, odnoszącą się do oprogramowania komputerowego. Potrafi ono, niezależnie lub z pomocą ludzi, tworzyć pewną zawartość gry. Zważywszy na to jak młoda jest to dyscyplina, formalna definicja nie jest jeszcze zupełnie określona. Najczęściej przywoływana jest interpretacja PCG jako algorytmicznego tworzenia zawartości gier z ograniczonymi lub pośrednimi danymi wejściowymi użytkownika [23]. Pojęcie proceduralnego generowania odnosi się także do takich branż jak filmowa, muzyczna, czy graficzna [26].

2.1. Geneza pojęcia PCG

Za pierwsze zastosowanie proceduralnego generowania zawartości w branży wirtualnej rozrywki uznaje się rok 1980 oraz produkcję o tytule *Rogue* [11]. Ta prosta gra została stworzona m.in. przez Michaela Toya, Glenna Wichmana i Kena Arnolda. Za interfejs graficzny miała wyłącznie siedmiobitowy system kodowania znaków (ang. ASCII – American Standard Code for Information Interchange), a polegała na przemierzaniu korytarzy oraz pomieszczeń lochów w poszukiwaniu legendarnego *Amuletu Yendoru*. Tytuł ten zapoczątkował cały podgatunek gier fabularnych (ang. RPG – role–playing game) nazywanych odtąd *roguelike*. Charakterystycznym dla nich elementem jest losowość przedstawionego świata. Układ pokoi, jak również znajdujący się w nich przeciwnicy i skarby są ustalane losowo i unikalnie dla każdej rozgrywki. Rozgrywka kończy się wraz z tzw. trwałą śmiercią postaci (ang. permadeath). Przykładowa struktura lochów przedstawiona jest na rys. 2.1.

Bezpośrednią motywacją do zastosowania w *Rogue* wspomnianej losowości według Glenna Wichmana była idea stworzenia gry nieprzewidywalnej. Takiej, która byłaby niespodzianką dla samych autorów [25]. W tamtych czasach wszelkie gry



Rysunek 2.1: Pokazowy obraz z gry *Rogue*.

przygodowe, miały dokładnie taką samą strukturę przy każdym uruchomieniu. Schemat fabuły w nich zawartej był stały, a co za tym idzie, to sami twórcy musieli ją wymyślać na etapie produkcji.

Niedługo po dołączeniu *Rogue*'a do popularnej w tamtych czasach wersji platformy komputerowej *UNIX* – Barkley Standard Distribution 4.2 (*BSD*), świat ujrzał produkcję wydaną przez *Acornsoft* na komputery *BBC Micro* i *Acorn Electron* [7], która również przeszła do historii jako rewolucyjna. W grze *Elite* gracz wciela się w rolę w kapitana statku kosmicznego. Oprócz samego przemierzania przestrzeni kosmicznej rozmaitych galaktyk, można przewozić i handlować towarami, wykonywać misje wojskowe, poszukiwać nowych surowców lub być kosmicznym piratem atakując inne statki kosmiczne. Twórcy gry, David Braben oraz Ian Bell, również skorzystali z możliwości generowania treści, jednak na dużo większą skalę i z zupełnie innych pobudek niż miało to miejsce w *Rogue*. Obie produkcje były tworzone w latach osiemdziesiątych ubiegłego wieku. Platformy komputerowe, na które powstawały nowe produkcje miały wtedy bardzo ograniczoną pamięć. Autorzy *Elite* mieli do dyspozycji zaledwie 22kB, lecz nie przeszkodziło im to w stworzeniu pierwszej gry komputerowej z grafiką symulującą trzeci wymiar (dokładniej mówiąc krawędziowo modelowaną w przestrzeni 3d, ang. wire-frame). Na rys. 2.2 zaprezentowana jest pseudo–trójwymiarowa grafika statku ukazująca się użytkownikowi po uruchomieniu programu.

Osiem galaktyk, każda z różną liczbą planet, których cechy (nazwa, położenie, ceny towarów, stopień i rodzaj zaludnienia czy nawet same opisy) również były unikatowe. To wszystko mogło być osiągnięte tylko przez generowanie ich w sposób proceduralny. Braben i Bell na potrzeby swojej produkcji zaimplementowali deterministyczny generator liczb pseudolosowych, który przy pomocy *ziarna* (ang. seed) przypisywanego podczas rozpoczęcia każdej nowej rozgrywki, budował wyżej wymienione elementy świata przedstawionego. Autorzy, nie mogąc zmieścić wszystkich

Rysunek 2.2: Przykładowy obraz z menu startowego gry *Elite*.

danych bezpośrednio w kodzie źródłowym, zdecydowali się na ich proceduralny opis.

Są to więc dwa przykłady pierwszych zastosowań PCG z wczesnych lat 80–tych, oba jednak stosujące tę technikę w różnych celach. *Rogue* skupia się na unikalności doznania i zwiększeniu tzw. *imersji* (ang. immersion, zanurzenie się w świecie przedstawionym, np. w grze lub książce). Natomiast *Elite* przy jednoczesnym odpowiedzeniu na te same problemy, koncentruje się dodatkowo na skali przedstawionego świata, starając się tym samym pokonać ograniczenia sprzętowe ówczesnych urządzeń. Blisko cztery dekady później producenci gier wciąż sięgają do korzeni branży, czerpiąc z nich inspiracje i stosują proceduralny sposób opisu treści także w najnowszych dziełach.

2.2. Przykłady generowanych treści

Na przestrzeni niemal czterdziestoletniej historii PCG powstało wiele gier aplikujących omawianą metodologię na rozmaite sposoby. Według Togeliusa i wsp. w [20]

metody proceduralnego tworzenia zawartości gier możemy łatwo sklasyfikować według czasu, w którym działają lub ich roli w grze, zakresie kontroli, technik konstrukcji czy też stopniu generyczności oraz losowości zastosowanych rozwiązań, a także po stopniu, w jakim angażują użytkownika.

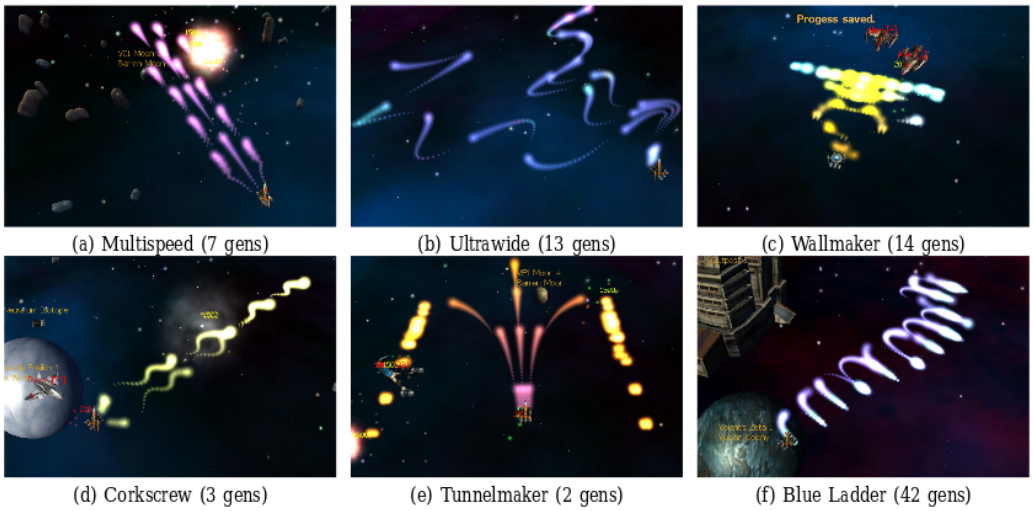
Przede wszystkim nie zawsze wygenerowane obiekty są niezbędnymi, integralnymi elementami świata gry. Przykładem może być generowanie broni, gdzie nazwa czy wygląd nie wpływają bezpośrednio na rozgrywkę, a co najwyżej na odczucia gracza. Z drugiej strony mamy treści będące absolutnie krytyczne dla gry. To mogą być nawet całe poziomy plansz, jak ma to miejsce w *Super Mario Bros*, *Rogue* czy *Spelunky*, będące produktami, w którym PCG stanowi ich absolutny trzon i esencję. Źle wykonane, wpływają bezpośrednio na przebieg gry, np. uniemożliwiając ukończenie danego poziomu lub wprost przeciwnie – upraszczając go do tego stopnia, że przejście go nie dostarcza użytkownikowi żadnej rozrywki.

Techniki PCG mogą być stosowane online, jak ma to miejsce np. w grze *Left 4 Dead*, która potrafi dostosować poziom rozgrywki adekwatnie do dotychczasowych wyników gracza lub offline, jak np. w *Forza Motorsport*, gdzie wytrenowany offline automat może zastępować gracza w wyścigu oddając jednocześnie jego styl jazdy.

Zgłębiając dalej temat klasyfikacji należy wspomnieć o sposobach kontrolowania wyników pracy algorytmów PCG. Jednym ze sposobów osiągnięcia takiej kontroli jest użycie seed'a. Pozwala on na całkowite odtworzenie losowo wygenerowanych struktur, a co za tym idzie, kontrolowanie ich. Ma to miejsce np. w przypadku gry *Minecraft*, gdzie zbudowany w ten sposób został cały wirtualny świat. Innym sposobem uzyskania kontroli nad efektami pracy algorytmów, który został zastosowany m.in. w *Infinite Mario Bros*, jest tzw. wektor cech (ang. features vector). W przytoczonym przykładzie zbiór parametrów odpowiadający za charakter generowanej planszy składa się z sześciu liczb całkowitych: liczba przerw między platformami, ich średnia szerokość, liczba przeciwników i ich położenie wyrażone za pomocą trzech prawdopodobieństw, liczba występujących ulepszeń (ang. powerups) oraz liczba różnych rodzajów skrzyń [19].

Przyjęło się, że generatory działają niezależnie, a gracz jest tylko odbiorcą ich pracy. Jednak relacja użytkownika z grą bywa coraz częściej symetryczna. Algorytmy zaczynają korzystać również z informacji dostarczanych przez gracza do wirtualnego świata. Zdolności adaptacyjne PCG bazują głównie na interakcji osoby z aplikacją. Analizując sposób zachowania gracza, generator potrafi dopasować strategię lub odpowiednio modelować treść gry. Choć dotychczas w przemyśle dominowało podejście generyczne – niebiorące pod uwagę poczynań gracza – a adaptacyjne PCG było raczej tematem do akademickich rozważań, to warto podkreślić, że ta sytuacja zaczyna się zmieniać. Obecnie wiele gier implementuje środowisko reagujące na odczucia użytkownika lub odpowiadające na potrzeby odbiorcy, dając mu więcej obiektów takiego rodzaju, którego on dotychczas preferował. Ma to miejsce np. w produkcji *Left 4 Dead*, gdzie twórcy stworzyli system balansujący intensywność

wywoływanych u gracza emocji [4]. Takie podejście zastosowano również w grze *Galactic Arms Race*, gdzie proponowane graczowi bronie są generowane przez algorytm cgNEAT (content–generating NeuroEvolution of Augmenting Topologies) na podstawie jego poprzednich wyborów i dotychczasowych preferencji [9]. Poglądowe rezultaty algorytmu zostały zaprezentowane na rys. 2.3.



(a) Multispeed (7 gens) (b) Ultrawide (13 gens) (c) Wallmaker (14 gens)

(d) Corkscrew (3 gens) (e) Tunnelmaker (2 gens) (f) Blue Ladder (42 gens)

Rysunek 2.3: Przykłady wygenerowanych sposobów działania broni wygenerowanych przez cgNEAT w grze *Galactic Arms Race* [9].

Tematu losowości w przypadku schematycznego generowania treści nie da się pominąć. Tu również można znaleźć różne podejścia, według których dokonuje się podziału. Determinizm w PCG pozwala nam, przy zachowaniu tych samych założeń startowych i parametrów, na ponowne generowanie tej samej zawartości. Takie podejście wykorzystano we wspomnianej wcześniej grze *Elite*, gdzie na deterministycznym silniku liczb pseudolosowych oparto całą grę. Nie zawsze jest to jednak potrzebne, a czasem nawet bywa niepożądane. Zdarza się, że naszym celem jest zapewnienie różnorodności i nie zależy nam na odtwarzalności.

Sposób, w jaki tworzona jest treść, również należy do czynników, które podlegają klasyfikacji. Niektóre algorytmy PCG działają strukturalnie, jak np. wiele gier typu *roguelike*. Jednak istnieją również techniki polegające na wielokrotnym generowaniu i późniejszym testowaniu rezultatów w pętli aż do otrzymania satysfakcjonującego wyniku. Ten typ nazywany jest „generuj i testuj” (ang. generate–and–test). Tak powstał np. *Yavalath*– gra planszowa całkowicie wygenerowana przez program komputerowy właśnie w paradygmacie generate–and–test przy użyciu algorytmów ewolucyjnych [2].

Jednak propozycji zdefiniowania zbioru kategorii, który miałby pomóc w klasyfikacji PCG było więcej. Za przykład może posłużyć jeszcze Hendrikx i wsp., którzy w [11] podzielili proceduralne generowanie zawartości na cztery grupy:

1. Fragmenty gry (ang. game bits)– np. tekstury, dźwięk;

2. Przestrzeń gry (ang. game space)– przedstawiony świat gry;
3. Systemy gry (ang. game systems)– mechanika gry, złożone relacje między obiektami;
4. Scenariusze gry (ang. game scenarios) – np. fabuła, treści zadań.

Według tej klasyfikacji gra *Elite* zalicza się do grupy drugiej, trzeciej i czwartej. *Rogue* natomiast zawiera się w grupie drugiej oraz czwartej. Jako przykład dla grupy pierwszej może posłużyć gra pt. *.kkrieger*. Prosta z pozoru gra pierwszoosobowa studia *.theprodukkt* z 2004 roku, ważąca zaledwie 96kB generowała tekstury, modele obiektów, muzykę i dźwięki. To właśnie niewielki rozmiar programu sprawił, że w kontekście prezentowanych możliwości została ona doceniona na arenie międzynarodowej.

2.3. Wady zastosowania PCG

Pomyślne zaprojektowanie generatora treści bywa nie lada wyzwaniem. Często to, co z jednej strony uznawane za zaletę, poprowadzone do ekstremum może okazać się wadą takiego rozwiązania. W przypadku treści podstawowej, jak w przypadku *Infinite Mario Bros*, PCG może np. utrudnić rozgrywkę do nieludzkiego niemal poziomu lub skrajnie ją ułatwić, pozbawiając gracza wyzwania. Generatory, które tworzą całe zasady gier planszowych takich jak np. *Yavalath* od *LUDI* [2] czy *Turncoat–Chess* w [17], ryzykują wygenerowaniem takich, które okazują się niemożliwe do ukończenia lub takich, których zasady są zbyt skomplikowane dla ludzkiego gracza.

Dla strategii czasu rzeczywistego (ang. RTS – Real Time Strategy) jak np. *Starcraft*, istotną cechą każdej planszy jest jej zbalansowanie. Fakt, że przeciwnik będzie miał do przebycia krótszą drogę ze swojej lokacji startowej do danego źródła zasobów, może okazać się kluczowym czynnikiem ważącym o jego wygranej. Dodatkowo to zbalansowanie nieraz bywa nieoczywiste – niejednorodne. Jak w przypadku kampanii fabularnych, kiedy projektanci na potrzeby wymyślonej wcześniej historii potrzebują stworzyć mapę spełniającą konkretne wytyczne (rozmieszczenie przeciwników, ukształtowanie terenu etc.). Generatory takich kompletnych map byłyby bardzo ciężkie do kontrolowania[24].

Wspomniany wcześniej *.kkrieger* osiągał minimalną ilość pamięci, generując niemal wszystko, co możliwe. Przypłacił to jednak czasem ładowania, który potrafi trwać aż do pół godziny.

W przypadku próby urozmaicenia przez PCG elementów pobocznych świata gry, jak drzewa, ogień, przeciwnicy, bronie czy nawet same zadania, twórcy dążą do uzyskania wspomnianego wcześniej efektu imersji, nadając tym obiektom unikalności i naturalności. Niestety wiele produkcji minęło się z tym celem, idąc w przeciwnym

kierunku i osiągając raczej schematyczność i wrażenie sztuczności. Ostatecznie według Elizabeth Reid w [18] światy wirtualne nie istnieją tylko dzięki technice, użytej do ich reprezentacji, ani nie wyłącznie w umyśle użytkownika, ale w relacji tych konstrukcji. Iluzja rzeczywistości nie spoczywa w samej aparaturze, lecz w chęci użytkowników, by wytwory ich wyobraźni traktować tak jakby były rzeczywiste. Dlatego też osoby odpowiedzialne za projektowanie świata gry często ponoszą fiasko. Przykładem jakim można się posłużyć, jest *The Elder Scrolls V: Skyrim*, gdzie chciano osiągnąć wrażenie niepowtarzalności, generując całe zadania poboczne. Dziś jednak uznawane są one na ogół jako nudne, bez wyrazu i pozbawione większej głębi.

2.4. PCG a labirynty i poziomy w grach

Labirynt – skomplikowana konstrukcja, mająca pierwotnie swoją złożonością zwodzić nieproszonych gości królewskich grobowców lub jako nietrywialne figury geometryczne zdobić prace ówczesnych artystów. Obecnie raczej kojarzy się nam z ćwiczeniami dla dzieci lub zagadkami logicznymi dla dorosłych. W erze digitalizacji nie trzeba budować architektonicznie skomplikowanych budowli, których urzeczywistnianie trwa latami. Dzięki komputerom można prezentować te struktury wirtualnie, np. w grach, gdzie forma labiryntu znajduje swe zastosowanie w wydłużeniu i urozmaiceniu rozgrywki, a czasem jest motywem przewodnim. Jednak jeden problem pozostaje uniwersalny niezależnie od epoki i technologii w niej dostępnej. Każda z takich struktur musi być najpierw zaprojektowana. Bez względu na to, czy tworzony jest fizyczny labirynt, czy też wirtualny, potrzebna jest osoba, która go zaplanuje. PCG wydaje się być odpowiedzią na ten problem, którego pozornie bez bezpośredniego udziału człowieka nie da się rozwiązać. W grach komputerowych labirynty przyjmują różne postacie.

2.4.1. Przykłady labiryntów w grach

Lochy, jaskinie i inne tym podobne struktury w grach są według Noor Shaker i wsp. w [20] środowiskiem, gdzie gracz w jednym punkcie wchodzi, gromadzi skarby, unika lub zabija potwory i ostatecznie wychodzi w innym. Taki schemat można spotkać choćby w opisanym na początku rozdziału 2.1 *Rogue* z 1980 r. To właśnie dzięki nieprzewidywalnym labiryntom dostarczał on rozrywki i nieoczekiwanych przygód. Jednym ze spadkobierców przytaczanego klasyka i reprezentantem gatunku *roguelike* jest *Diablo* – seria gier, która na nowo zdefiniowała ten rodzaj. Ubiera ona rozgrywkę w dopracowany interfejs graficzny, który zdecydowanie bardziej niż ten tekstowy oddaje naturę podziemi jako zimnego, mrocznego i przerażającego miejsca. Taki obraz lochów można znaleźć niemal w każdym komputerowym RPG (np. seria *Legends of Zelda*, *Final Fantasy*, czy *The Elder Scrolls V: Skyrim*), jednak jego początków należy się dopatrywać w *Dungeons and Dragons*, znanej grze fabularnej z 1974 r. W większości tzw. przygodówek struktura lochów składa się z pokoi połączonych

ze sobą korytarzami. Choć pierwotnie termin „loch" odnosił się właśnie do miejsca pełnego więziennych cel, to w grach zwykł on przybierać również postać jaskiń, grot, a także rozmaitego rodzaju struktur architektonicznych. Co ciekawe, pomimo faktu, że rozgrywka gier RPG ma zupełnie inny charakter od tych platformowych, to istnieją wyraźne podobieństwa pomiędzy generowaniem poziomów i plansz dla obu tych typów. Jako przykład przytoczyć można *Spelunky*, w których występują takie charakterystyczne cechy z plansz gier przygodowych jak wolna przestrzeń ograniczona ścianami, skarby oraz przedmioty pomocne w pokonywaniu przeciwników i pułapek. To, co odróżnia te dwa gatunki od siebie, to przede wszystkim mechanika. W tzw. platformówkach gracz jest ograniczony przez grawitację, a jego agent może poruszać się tylko na boki i skakać na małe odległości. Tymczasem w RPG-ach użytkownik zazwyczaj nie jest ograniczony przez zorientowanie rozgrywki wyłącznie horyzontalne lub wertykalne.

2.4.2. Wybrane sposoby tworzenia labiryntów

Space partitioning

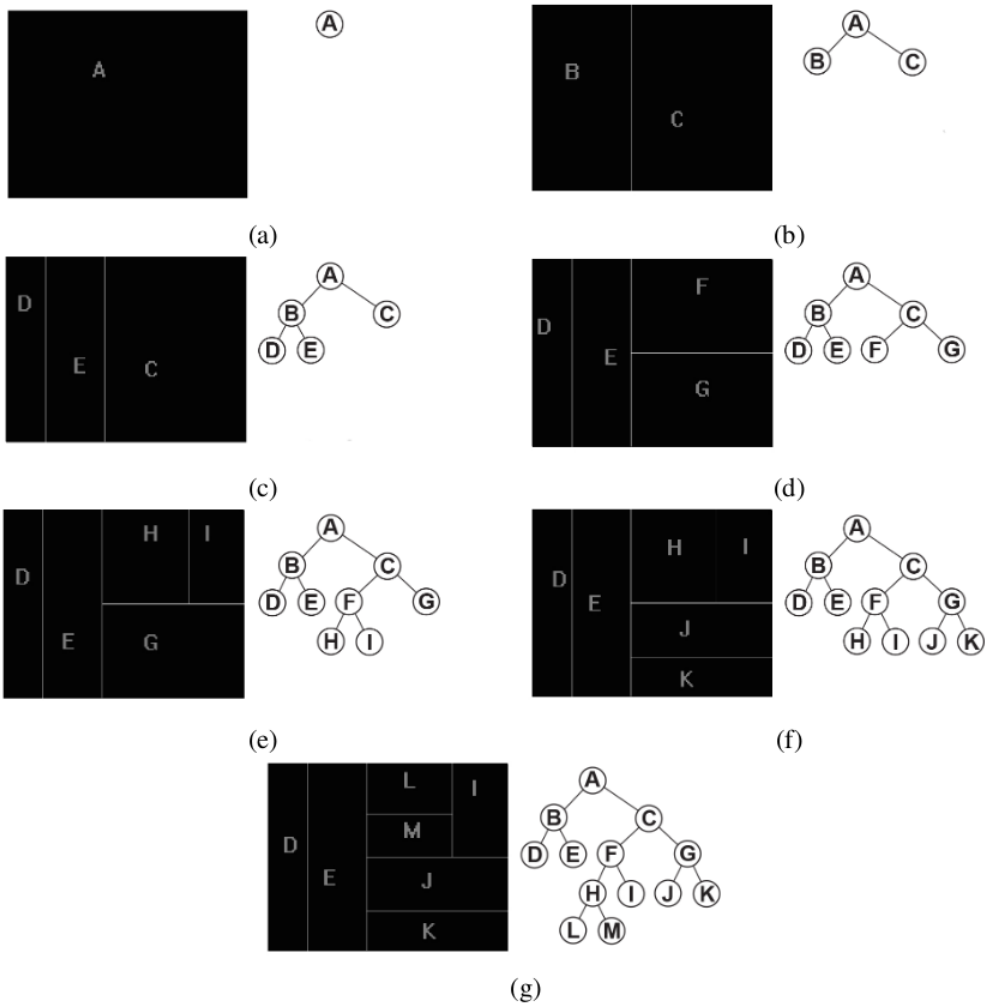
Pierwsza, bardzo prosta metoda generowania labiryntów jest oparta na binarnym partycjonowaniu przestrzeni (ang. BSP – binary space partitioning). Polega ona na rekursywnym dzieleniu obszaru na dwie części i następnie decydowaniu o zawartości sekcji (np. 1 oznacza pomieszczenie, a 0 to brak pomieszczenia), by ostatecznie połączyć ze sobą powstałe pokoje. To tworzy bardzo symetryczne i hierarchiczne poziomy. Algorytm działa w następujący sposób:

1. Rozważmy prostokątny obszar o szerokości s i długość d jako korzeń drzewa BSP.

2. Dzielimy przestrzeń poziomo lub pionowo na dwie niekoniecznie równe części, tworząc jednocześnie dwa kolejne węzły drzewa, które obecnie pozostają liśćmi.(rys. 2.4)

3. Liście ponownie podlegają rekurencji, chyba że ich szerokość lub długość są mniejsze od wyznaczonej minimalnej wartości (np. $s/4$ i $d/4$). (rys. 2.5)

Na końcu każdemu obszarowi zostaje przypisany pokój mieszczący się w jego granicach. W tym celu losowane są dwa punkty będące odpowiednio lewym górnym oraz prawym dolnym rogiem nowego pomieszczenia. Należy sprawdzić, czy jego obszar jest zgodny z minimalnym akceptowalnym rozmiarem. Kiedy już takie drzewo zostanie wygenerowane, można przejść do ostatniego kroku, jakim jest łączenie powstałych pomieszczeń, przez poprowadzenie korytarza pomiędzy dziećmi tego samego rodzica.

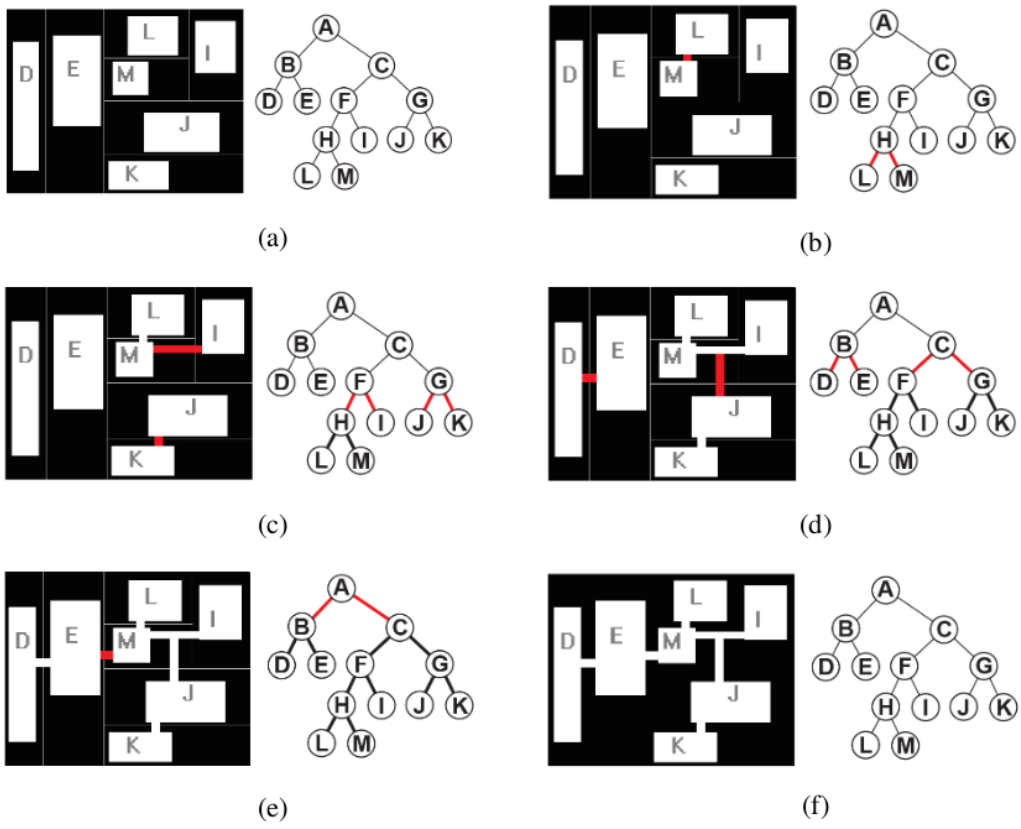


Rysunek 2.4: Przykład binarnego partycjonowania przestrzeni [20].

Technika ta i gwarantowana przez nią hierarchiczność łączenia pomieszczeń zapobiega również niechcianemu pokrywaniu się korytarzy, a także dostarcza pogrupowane pomieszczenia, dzięki czemu każda z tych grup może posiadać jakąś inną cechę.

Agent–based

Generowanie labiryntu oparte na agentach, w przeciwieństwie do poprzedniej metody, traktuje temat raczej w skali mikro. W kontraście do usystematyzowanych, symetrycznych i poukładanych pomieszczeń z dzielenia przestrzeni, tworzy przy tym bardziej chaotyczne i mniej kontrolowane lochy. Bardzo ciężko przewidzieć wpływ zmian w parametrach odpowiadających za zachowanie sztucznej inteligencji (ang. artificial intelligence), dlatego znalezienie optymalnej konfiguracji wymaga sporego nakładu pracy przez testowanie. Do listy problemów należy jeszcze dodać brak pewności, że wygenerowane przez agenta pokoje nie będą się pokrywać czy, że będą



Rysunek 2.5: Przykład łączenia wcześniej podzielonej przestrzeni [20].

posiadały jakąś interesującą strukturę. Wariant tej metody, gdzie bot podejmuje swoje decyzje losowo, wyłącznie na podstawie swoich dotychczasowych ruchów, nazywany jest ślepym (ang. blind). „Blind" agent zaczyna w pewnym losowym punkcie przyszłego lochu, który pierwotnie jest obszarem wypełnionym komórkami o wartości ściany (wyrażonych np. jako „1") i również losowo jest wybierany początkowy kierunek. Szansa na zmianę kierunku czy na stworzenie nowego pokoju, przez takiego bota, który dopiero rozpoczyna swoją podróż, wynoszą 0%. Porusza się w zadanym na starcie kierunku zamieniając każdą z pozycji, na której stanął na *podłogę* korytarza (reprezentowaną np. jako 0). Jednocześnie z każdym krokiem w tym samym kierunku zwiększa się prawdopodobieństwo (np. o 5%) na zmianę tego kierunku oraz z każdym wykonanym krokiem przez agenta, w którym nie doszło do stworzenia pomieszczenia, również zwiększa się szansa na powstanie losowych rozmiarów pokoju (np. o 5%). Bot wykonuje zadany algorytm, dopóki utworzony loch nie jest wystarczająco duży.

Powyższy sposób budowania podziemi może się okazać użyteczny i skuteczny dla wybranych celów, niestety generuje on sporo problemów, jak możliwość nachodzenia na siebie pokoi, czy ślepe korytarze. By uniknąć takich problemów, możemy zmusić naszego agenta do „patrzenia w przód" (ang. look–ahead agent) na otoczenie.

Podobnie jak poprzedni „ślepy" bot, tak i ten niech rozpoczyna swoje działanie w losowym miejscu na mapie. Na początku sprawdza, czy dodanie nowego pokoju będzie kolidować z którymś z już istniejących. Jeśli dla każdej z możliwych wielkości pokojów występuje nakładanie się obszarów pomieszczeń, sprawdza, czy poprowadzenie korytarza spowoduje przecięcie się z jakimś z istniejących pokojów. Algorytm zatrzymuje się, w przypadku gdy agent znalazł się w miejscu, w którym nie może powstać żaden nowy pokój o dopuszczalnym rozmiarze oraz dowolnej dopuszczalnej długości korytarz, w dowolnym kierunku nachodzi na obszar zajęty przez pomieszczenie.

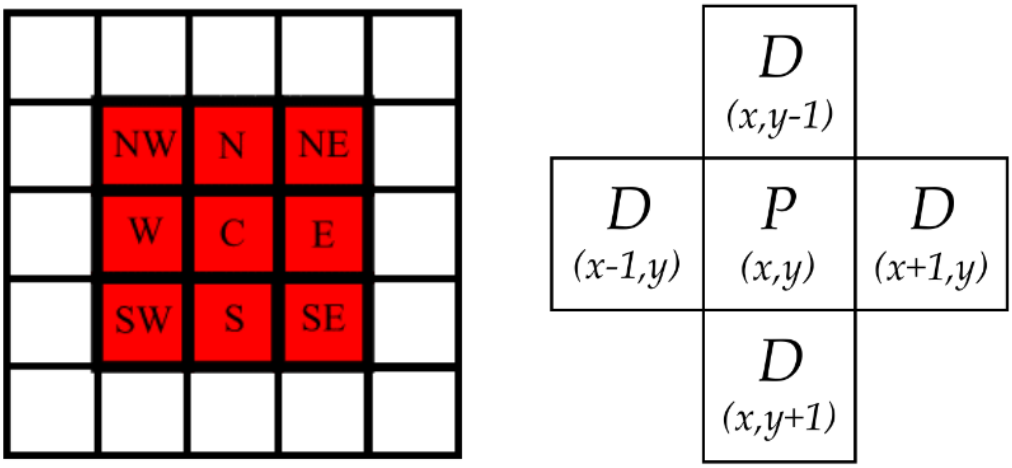
W obu tych przypadkach spotykamy się z pewnymi problemami. W pierwszym był to chaotyczny ruch bota, który może wracać po dopiero zbudowanej ścieżce, tworzyć nowe pokoje powielające te istniejące, co w rezultacie może skutkować mało interesującą planszą. Drugi bot, choć zapobiega błędom poprzednika, może prowadzić do bardzo szybkiego zakończenia generowania labiryntu, przez co możemy otrzymać strukturę co prawda bardziej przypominającą loch, lecz o bardzo ograniczonej długości.

Oczywiście oba z powyższych przykładów –„blind" i „look–ahead"– dzięki mniejszym lub większym usprawnieniom mogą zyskać na efektywności w generowaniu poziomów, jednak wymaga to wielu testów metodą prób i błędów. Choć dzięki wspomnianej chaotyczności i nieprzewidywalności algorytmu, finalne rezultaty mogą okazać się bardziej realistycznymi i naturalnymi podziemiami, to ryzykujemy generowaniem niemożliwych do ukończenia lub nieatrakcyjnych map.

Cellular automata

Automaty komórkowy (ang. CA – cellular automata) w szczegółach zostaną opisane w kolejnym rozdziale niniejszej pracy. Na potrzeby tego podrozdziału wystarczy nam wiedzieć, że cellular automaton składa się z n–wymiarowej siatki stanów (zazwyczaj dwu–wymiarowej w formie macierzy) oraz zbioru zasad przejść między nimi. W najprostszym przypadku zbiór stanów zawiera tylko dwa– 1, gdy komórka jest aktywna i 0 w przeciwnym przypadku. Sposób rozdystrybuowania stanów na siatce, na początku (przed pierwszą iteracją algorytmu) jest określany mianem stanu początkowego automatu komórkowego. Podczas każdej iteracji algorytmu każda z komórek decyduje o swoim nowym stanie na podstawie wspomnianego zbioru zasad odnoszącego się do obecnego stanu jej oraz jej „sąsiedztwa". Na potrzeby generowania podziemi interesuje nas najbardziej wersja dwu–wymiarowa automatu komórkowego, dla której najpopularniejsze dwa typy sąsiedztwa to według Moore'a i von Neumanna. Oba z nich mogą mieć dowolny naturalny rozmiaru wynoszący co najmniej jeden.

Sąsiedztwo Moore'a wielkości 1 dla danej komórki k można opisać jako kwadrat 8 komórek otaczających k w centrum (zaprezentowane na powyższej grafice po lewej stronie). Sąsiedztwo von Neumanna w wariancie o wielkości 1 wyglądem przypomina



Rysunek 2.6: Od lewej sąsiedztwo wg. Moore'a[1] oraz sąsiedztwo wg.Neumanna[2]

krzyż i składa się tylko z 4 komórek (odpowiednio po jednej nad, pod, z prawej i z lewej strony, zaprezentowane na powyższej grafice po prawej stronie).
Prezentowana przez Johnsona i wsp. w [12] metoda używa tylko czterech parametrów do kontrolowania rezultatu:

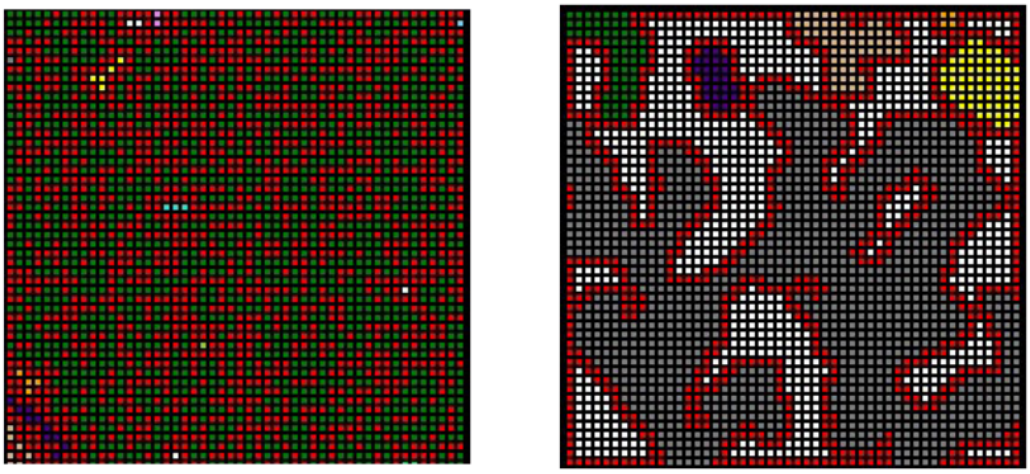
- procent (r) komórek, które pierwotnie posiadają stan aktywny (są *skałami*),
- liczbę iteracji automatu komórkowego (n),
- rozmiar sąsiedztwa (M),
- wartość graniczna, która definiuje komórkę jako *skałę* (T).

Autorzy publikacji założyli, że każdy pokój będzie siatką o wymiarach 50x50, moc zbioru stanów będzie wynosiła 2, a plansza będzie początkowo wypełniona komórkami nieaktywnymi (*podłogą*). Następnie Johanson i wsp. proponują następujący algorytm:

1. Na siatce zostaje rozdystrybuowany losowo szum w postaci komórek, których stan zmieniany jest na *skałę*,
2. Wykonanych zostaje n (np. 2) iteracji automatu komórkowego, którego działanie opisane jest za pomocą pojedynczej reguły, mówiącej, że komórka jest *skałą* wtedy i tylko wtedy, gdy liczba komórek sąsiadujących z nią w sensie sąsiedztwa Moore'a o stopniu M (np. 1), które są *skałami* wynosi co najmniej T (np. 5), w przeciwnym wypadku komórka jest pusta.

Dzięki tej prostej procedurze autorzy publikacji byli w stanie generować zaskakująco skutecznie i szybko lochy które przypominają naturalne podziemia (rys. 2.7)

[1]Sąsiedztwo Moore'a: `https://en.wikipedia.org/wiki/Moore_neighborhood`
[2]Sąsiedztwo von Neumanna: `https://en.wikipedia.org/wiki/Von_Neumann_neighborhood`



Rysunek 2.7: Powyższa grafika prezentuje porównanie siatki o losowo rozdystrybuowanych *skałach* oraz takiej na której przeprowadzono kilka iteracji CA (parametry to $r = 0.5$, $n = 4$, $M = 1$, $T = 5$)[12].

Metoda ta zyskuje na swojej atrakcyjności, gdy zwrócimy uwagę na małą liczbę parametrów potrzebnych do dostrojenia oraz gdy rozpatrzymy możliwe zastosowania. Jedno z nich to tworzenie „nieskończonych" jaskiń (teoretycznie, bo oczywiście w praktyce żadna jednostka obliczeniowa nie jest w stanie przechować nieskończonej liczby informacji), które byłyby generowane dynamicznie w trakcie rozgrywki. Niewątpliwą wadą tej metody jest stosunkowy brak kontroli nad efektami końcowymi. Niełatwe okazuje się odgadnięcie jakie parametry zaskutkują jaką charakterystyką lochu. Przykładowo czy po zmianie danego parametru będzie więcej tuneli, czy więcej rozłącznych nieosiągalnych pokoi z tego powodu i ta metoda wymaga wielu testów metodą prób i błędów.

Rozdział 3.

Zastosowane metody

W tym rozdziale omówione zostały *automaty komórkowe* oraz *algorytmy ewolucyjne* (ang. EA – evolutionary algorithms). Zarówno CA, jak i EA znajdują szerokie zastosowanie w PCG. Metody te, choć często stosowane niezależnie, mogą być również łączone, jak ma to miejsce np. w [15]. Mitchell i wsp., dzięki wykorzystaniu algorytmów ewolucyjnych, odkryli nowe zasady poprawiające rezultaty CA w klasyfikacji gęstości i synchronizacji komórek.

3.1. Automaty komórkowe

Ojcem automatów komórkowych jest węgierski uczony Janos Lajos Neumann, który chciał za ich pomocą opracować model maszyny zdolnej do samoreprodukcji, tzn. powielania swojej struktury. Później koncepcje Neumanna uprościł nieco Edgar Frank Codd, przybliżając ją do modelu, jaki obecnie utożsamiamy z nazwą automatu komórkowego, którego można postrzegać jako formę siatki jednakowych oddziałujących na siebie nawzajem automatów zwanych właśnie komórkami (ang. cells). Cellular automaton charakteryzuje się:

- n-wymiarową siatką komórek,
- zbiorem S dostępnych dla każdej komórki stanów,
- zbiorem reguł przejść między tymi stanami, które odnoszą się do wartości komórki i jej sąsiedztwa.

Podczas każdej iteracji algorytmu wyliczany jest stan danej komórki w oparciu o stany komórek w jej sąsiedztwie oraz wspomniany zestaw reguł przejść między stanami. Dopiero gdy wszystkie komórki są obliczone, następuje podmiana wartości z tymi z poprzedniej iteracji. Powyższe cechy przypisywane są tzw. deterministycznemu automatowi komórkowemu. Gdy wspomniane reguły zależą od zmiennej losowej, wtedy mówi się o tzw. wersji probabilistycznej.

Choć idea samo-powielającej maszyny wciąż wydawała się być poza zasięgiem nowego modelu, to pozwalał on już na obliczenie każdej możliwej funkcji oraz mógł się reprodukować, co wystarczyło do skonstruowania dziś klasycznego przykładu automatu – tzw. Gry w życie (ang. Life). W 1970 r. John H. Conway w październikowym wydaniu „Scientific American" zaproponował automat komórkowy z dwuwymiarową siatką oraz określonym na niej sąsiedztwem Moore'a rozmiaru 1. Matematyk zaproponował dwuelementowy zbiór stanów oraz następujące zasady przejść:

- Każda martwa komórka posiadająca co najmniej 3 z 8 sąsiadów żywych – ożywa.
- Każda żywa komórka z 2 albo 3 żywymi sąsiadami – pozostaje żywa
- w przeciwnym wypadku umiera z osamotnienia lub zatłoczenia.

Automaty komórkowe znalazły swoje zastosowanie również w przemyśle gier komputerowych. Przykładem mogą być choćby procesy fizyczne jak ciepło i ogień, deszcz i zachowanie cieczy, czy ciśnienie i eksplozje [6]. Jak P. Sweetser i J. Wiles pokazali, CA w połączeniu z mapami wpływów (ang. influence maps) znacznie pomaga w podejmowaniu decyzji przez boty [21]. Natomiast w [16] CA odpowiada za modelowanie termicznej i hydraulicznej erozji w procesie generowania terenu. Tobias Mahlmann i wsp. w [13] przywołują automat komórkowy jako środek do generowania map w Dune 2 - jednej z pierwszych gier strategicznych czasu rzeczywistego. Również opisana już w rozdziale 2.4.2. praca Lawrence'a Johnsona i wsp. prezentuje metodę generowania „nieskończonych" poziomów jaskiń w czasie rzeczywistym właśnie za pomocą cellular automata [12]. Jak widać, metoda ta od dawna cieszy się dużym zainteresowaniem przemysłu gier i znajduje w nim szerokie zastosowanie. Tym istotniejszy staje się problem znajdowania odpowiednich parametrów dla automatów komórkowych w celu uzyskania satysfakcjonujących wyników. W tym celu stosuje się sieci neuronowe [8] oraz algorytmy ewolucyjne [1, 13] o których traktuje rozdział 3.2. niniejszej pracy.

3.2. Algorytmy ewolucyjne

Inspirowane naturalnymi procesami doboru naturalnego, algorytmy ewolucyjne należą do rodziny algorytmów optymalizacyjnych i są poddziedziną sztucznej inteligencji. Bazujący na darwinistycznej teorii selekcji naturalnej, niedeterministyczny algorytm przeszukuje przestrzeń alternatywnych rozwiązań danego problemu w poszukiwaniu grupy tych najlepszych. Zaczyna od wygenerowania populacji początkowej, tzn. zbioru elementów, nazywanych chromosomami, które podczas kolejnych iteracji będą modyfikowane. Każdorazowo funkcja ewaluacyjna ocenia każdego z kandydatów populacji, a na podstawie otrzymanych wyników, ze zbioru tego wydzielona

zostaje grupa osobników, które będą mogły kontynuować reprodukcję. Pozostałe elementy zostają odrzucone, jako skutek selekcji naturalnej. Jak łatwo można zauważyć, metoda ta jest radykalna, bowiem pozostaje niewrażliwa na różnice pomiędzy kolejnymi kandydatami wewnątrz rankingu. Na szczęście istnieją także selekcje wielokryterialne, korzystające z kilku funkcji oceny. Bez względu na sposób odrzucania słabszych chromosomów, rozradzanie się pozostałych, które przetrwały, technicznie polega na mieszaniu ich wzajemnie między sobą (przez kombinacje, krzyżowanie) lub kopiowaniu ich z małymi losowymi zmianami (mutacjami). Co ważne, pierwotna populacja może być absolutnie losowa, wypełniona całkowicie niewłaściwymi osobnikami, a dobrze zdefiniowana funkcja ewaluacyjna i tak rozpozna osobniki mniej niewłaściwe, tworząc z nich kolejne pokolenia.

Historia rozwoju tej dziedziny sięga lat 60tych ubiegłego wieku, kiedy to rozwijane były trzy, wtedy uważane za odrębne, nurty: pochodzące ze Stanów Zjednoczonych pojęcia „Programowanie ewolucyjne" i „Algorytmy genetyczne" oraz wywodzący się z Niemiec termin „Strategii ewolucyjnych". Dopiero w latach dziewięćdziesiątych zostały one ze sobą ściśle powiązane w jednym pojęciu „obliczeń ewolucyjnych". Co więcej, rozszerzono tę grupę o jeszcze jedną kategorię „programowania genetycznego". Dziś ten podział jest bardziej spójny. Programowanie genetyczne odnosi się do ewolucji struktur drzewiastych, najczęściej za pomocą operatorów zaburzających o losową wartość z rozkładu normalnego. Jest ono jedną z klas algorytmów ewolucyjnych, z których dwie wybrane zostały opisane poniżej.

3.2.1. Strategie ewolucyjne $\mu + \lambda$

Strategia ewolucyjna (ang. ES – evolution strategy) charakteryzuje się tym że podczas ewolucji populacji stosowana jest wyłącznie mutacja. Parametr λ odpowiada liczbie potomstwa generowanego podczas każdej iteracji, natomiast μ reprezentuje liczbę osobników mających każdorazowo przetrwać selekcję. Oto prosty przepis:
Populacja jest inicjalizowana $\mu + \lambda$ kandydatami. Następnie wykonywane są obliczenia ewaluacyjne, na podstawie których sortowany jest cały zbiór. Wtedy usuwane jest λ najsłabszych chromosomów, które zastępowane są przez λ zmutowanych kopii pozostałych przy życiu μ osobników. Algorytm choć prosty radzi sobie zaskakująco dobrze nawet w konfiguracji $(1 + 1)$-ES, gdzie tak naprawdę modyfikowany jest pojedynczy element, nie dopuszczając gorszych wyników niż obecny. Warto podkreślić, że techniki mutacji są ściśle powiązane ze sposobem reprezentacji kandydatów. Jeśli jest to np. wektor liczb rzeczywistych, dobrym sposobem na skuteczną mutację jest dodawanie losowej wartości
Pierwsze wykorzystanie ES to środek wspomnianych lat sześćdziesiątych, kiedy to znalazły zastosowanie w Berlinie podczas prowadzonych tam prac przez P. Bienerta, I. Rechenberga, i H.-P. Schwefela nad optymalnymi, pod względem oporu powietrza, kształtami ciał (pocisków) [3].

3.2.2. Algorytmy genetyczne

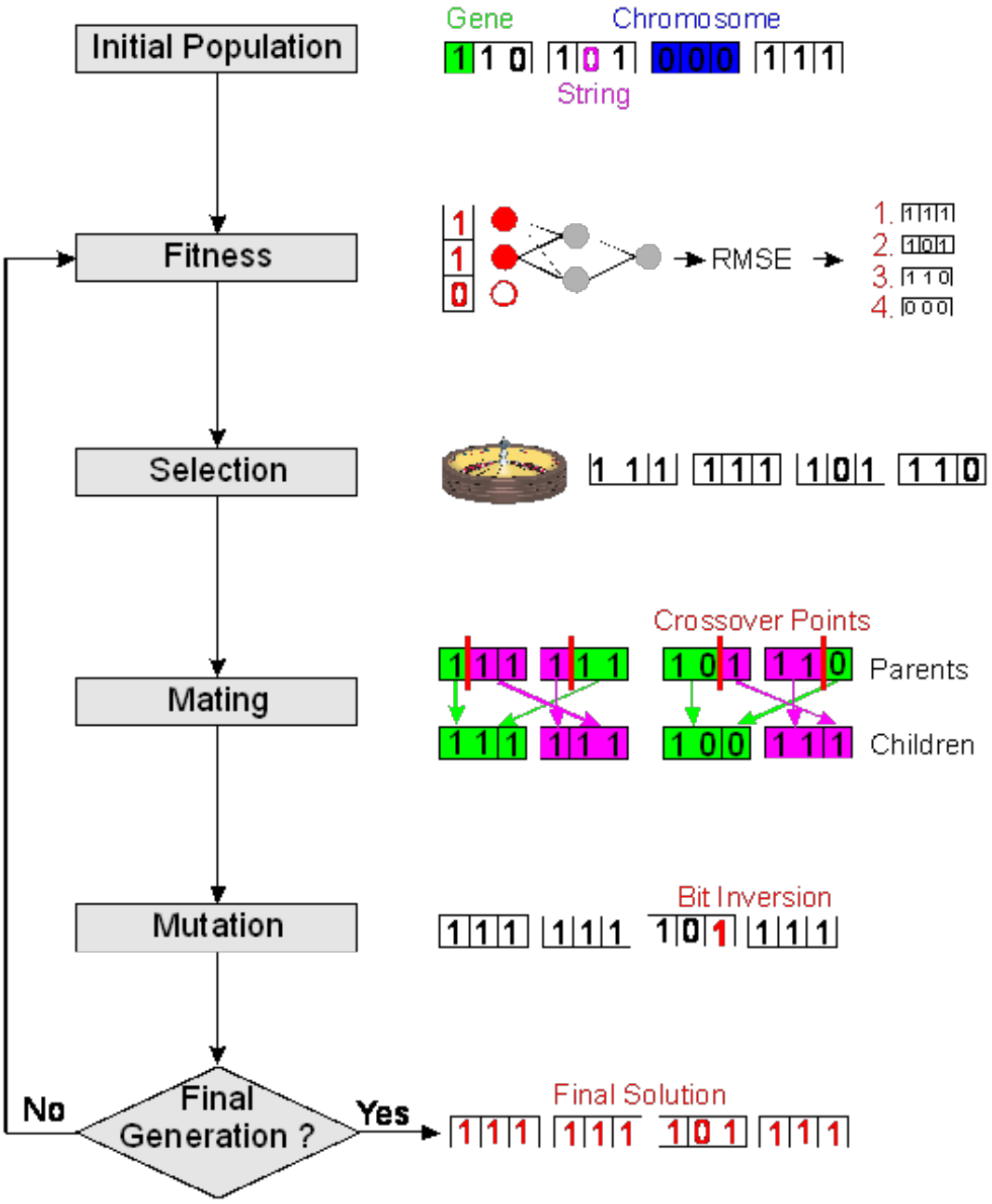
To rodzaj tylko pozornie bardziej skomplikowanej metaheurystyki od ES. Idea tego rozwiązania opiera się na łączeniu w losowy sposób genomów nieprzypadkowo wybranych chromosomów. Oprócz podstawowej mutacji nowe pokolenie powstaje na skutek krzyżowania się (ang. crossover) elity, która przetrwała ostatni krok selekcji.

Kojarzenie ma za zadanie stworzyć potomka o zespole cech będącym kombinacją cech rodziców. Rodzaj operatora krzyżowania, podobnie jak wcześniej w przypadku mutacji, ściśle zależy od reprezentacji chromosomu. Przykładowo czerpiąc inspirację z genetyki, możemy zastosować naturalny crossover, polegający na rozcinaniu dwóch chromosomów i sklejaniu nowego z pochodzących od nich różnych części. Taka metoda możliwa jest w przypadku, gdy reprezentujemy genomy np. za pomocą kodowania binarnego lub całkowitoliczbowego. Innymi sprawdzonymi operatorami krzyżowania są średnia wartość genów (w przypadku liczb rzeczywistych jako cech) czy też podstawowe operacje logiczne (dla kodowania binarnego).
Mutacja w algorytmach genetycznych (ang. GA – genetic algorithms), choć zdecydowanie pełni rolę drugoplanową, to wciąż ma ważne zadanie wprowadzania różnorodności do populacji. Mutacja zapobiega utrzymywaniu się w lokalnych ekstremach oraz dba o szerszą eksplorację przestrzeni dostępnych rozwiązań. Prawdopodobieństwo wystąpienia mutacji w GA jest zazwyczaj niskie (rzędu 1%), ponieważ intencją nie jest niszczyć już i tak dobre rozwiązania, tylko je różnicować.

Przykładowo, dla reprezentacji binarnej genomu, stosowane zazwyczaj operatory mutacji to losowanie i następnie zamiana dwóch bitów miejscami, lub negowanie losowo wybranego genu. Bezpiecznym operatorem mutacji dla chromosomów w postaci ciągu liczb całkowitych, jest prosta permutacja. W przypadku genomów przedstawianych liczbami rzeczywistymi sugerowane jest modyfikowanie losowo wybranych cech losowymi wartościami o standardowym rozkładzie normalnym

3.2.3. Funkcje ewaluacyjne

Sprawność każdego z omawianych dotychczas algorytmów ewolucyjnych opiera się na skutecznej funkcji oceny. Na podstawie ich działania dokonywana jest selekcja, która jest esencją mechanizmu ewolucji. Lecz odpowiednie ich dobranie jest niezwykle trudne i jest to główna część projektowania procesu, którego złożoność polega na subiektywności odczuć twórcy. W interesującym nas przypadku gier każdy z nas ma inne poczucie atrakcyjności, a rozrywka jest niezwykle trudna do zmierzenia a jeszcze trudniejsza do formalnego zdefiniowania. To charakter rozgrywki decyduje o sposobie, w jaki możemy mierzyć rozrywkowość jej elementów. Przykładem mogą być badania na temat atrakcyjności torów wyścigowych w grach samochodowych.
Funkcje ewaluacyjne można podzielić na 3 kategorie: bezpośrednie, symulacyjne oraz



Rysunek 3.1: Poglądowy schemat blokowy algorytmu genetycznego[1].

interaktywne.

Bezpośrednie ewaluacje bazują swoje obliczenia czysto na zawartości wygenerowanego elementu. Są one przez to bardzo szybkie i często łatwe do zaimplementowania. Dla przykładu taka funkcja może odnosić się do rozłożenia na mapie dostępnych dla gracza zasobów czy stosunku liczby jednych składowych do liczby drugich. Swoje działanie mogą opierać o teorie (wlicza się w to intuicja, rezultaty badań naukowych) albo o fakty i dane (ankiety, badania fizjologiczne użytkowników).

[1] Źródło: `http://www.frank-dieterle.com/phd/2_8_5.html`

Symulacyjne funkcje ewaluujące to kolejna rodzina funkcji ważących o losach populacji. Nazwa zdaje się dawać nam pewną intuicję, „symulacja" odbywa się za pomocą systemu agentów używających wygenerowanego obiektu, np. grających w stworzony poziom gry. Różne strategie wykorzystywane przez boty będą mierzyły różne właściwości dostarczonej przez dany zestaw cech zawartości. I tak jeśli chcemy zadbać o użytkowość treści np. grywalność, wtedy nasz automat powinien być skupiony na spełnieniu celu (np. na ukończeniu labiryntu czy poziomu gry 2D). W przypadku dbania o dostarczenie użytkownikowi pewnych wrażeń, chcielibyśmy agenta symulującego możliwie najdokładniej ludzkie zachowania (np. jak to jest prezentowane w [22], gdzie bazujące na sieciach neuronowych boty są trenowane, by jeździły jak prawdziwi gracze, a następnie używane są do ewaluowania generowanych torów wyścigowych). To podejście może być realizowane statycznie lub dynamicznie, tzn. agenci mogą zachowywać się w sposób deterministyczny, będąc specjalnie zaprojektowanymi pod konkretną rozgrywkę automatami lub też mogą znać jedynie podstawowe zasady, ucząc się pozostałych, jak i strategii podczas rozgrywki.

Ostatni wyróżniający się typ funkcji oceniających to **interaktywny**, który wymaga bezpośredniego wpływu użytkownika na proces. Może się to odbywać jawnie przez wyrażanie odczuć gracza w formie oceny np. toru wyścigowego lub niejawnie przez analizowanie jego zachowań jak np. intensywność korzystania z danego obiektu [10].

Rozdział 4.

Autorski program

W ramach pracy, zrealizowany został projekt programistyczny, implementujący, generator nieskończonych jaskiń wykorzystujący w tym celu automat komórkowy, który był inspirowany pracą Johnson i wsp. [12]. Na potrzebę osiągnięcia optymalnych wyników, na wybranych parametrach CA został zastosowany również algorytm genetyczny.

Generator zaimplementowany został w języku *Python 3* w paradygmacie obiektowym. Przyczyną takiej decyzji była wieloplatformowość języka oraz operujący na wysokim poziomie abstrakcji, intuicyjny sposób implementacji algorytmów.

4.1. Implementacja

Generowanie nieskończonego poziomu labiryntów zaczyna się od pojedynczego *kafelka* i przebiega w następujący sposób:

1. Inicjowany jest *kafelek* startowy ($(0, 0)$).
2. Kolejnym krokiem jest „odkrycie" go. W tym celu, o ile jeszcze nie zostali wygenerowani wcześniej, inicjowani są jego sąsiedzi (w rozumieniu sąsiedztwa 1-Moore'a) a potem następuje właściwa generacja jego samego. Oznacza to, że uruchomiona jest na nim określona liczba iteracji automatu komórkowego.
3. *Kafelek* centralny zostaje oznaczony jako „udostępniony" (ang. *accessed*) i jest zaprezentowany użytkownikowi.
4. Celem rozszerzenia mapy w którymś z 8 kierunków, wykonywane zostają kroki 2 i 3 dla leżącego tam obszaru.

Inicjowanie *kafelka* polega jedynie na samym utworzeniu obiektu oraz tzw. *zaszumieniu*, tj. przypisaniu losowo wybranym polom wartości początkowej. Automat

komórkowy uruchomiony podczas odkrywania obszaru korzysta z pól granicznych jego sąsiadów, dlatego niezbędne jest zainicjowanie ich przed próbą zatwierdzenia *kafelka*. Warto podkreślić, że wspomniane pola występują wyłącznie w trybie do odczytu, a zatem automat komórkowy nie nadpisuje wartości komórek z obszarów brzegowych sąsiadów.

Pojedynczy wyprodukowany obszar musi posiadać określone cechy, żeby móc go nazwać pomyślnie wygenerowanym:

1. *Kafelek* nie może być pokryty niemal wyłącznie (więcej niż 80%) komórkami jednej wartości.
2. Każdy z podobszarów wygenerowanego elementu, musi posiadać przejścia do co najmniej dwóch innych *kafelków*.

Te podstawowe właściwości nie gwarantują jeszcze potencjalnemu graczowi nieskończonej rozrywki. Do tego potrzebne jest zdefiniowanie dodatkowych własności, które jednak nie będą rozpatrywane w niniejszej pracy.

4.1.1. Reprezentacja automatu komórkowego

Do realizacji projektu zastosowano metodę automatu komórkowego, która została opisana w rozdziale 2.4.2. i 3.1. Podobnie jak w [12], do reprezentacji automatu komórkowego służy dwuwymiarowa siatka komórek o dowolnym rozmiarze, z dwuelementowym zbiorem stanów (*skała* lub *podłoga*).
Na ostateczny rezultat CA wpływają cztery parametry:

- INITIAL_RATIO (r) - liczba z zakresu $[0, 1]$, definiująca początkowy stosunek liczby komórek *skał* do komórek *podłogi*;
- NUMBER_OF_ITERATIONS (n) - dowolna liczba naturalna, oznaczająca liczbę iteracji automatu komórkowego;
- NEIGHBOURHOOD_SIZE (M) - rozmiar sąsiedztwa w rozumieniu Moore'a, wyrażony za pomocą dowolnej liczby naturalnej mniejszej od $1/2$ wielkości mapy (MAP_SIZE);
- CONDITIONS (C) - zbiór warunków granicznych dla każdego ze stopni sąsiedztwa, określający czy dane pole jest *skałą*, zdefiniowany jako lista $M + 1$ elementów, z których każdy jest wyrażony jako ciąg znaków (*string*) zawierający operator porównania i liczbę z zakresu $(0, 1)$ oddzielone pojedynczą spacją.

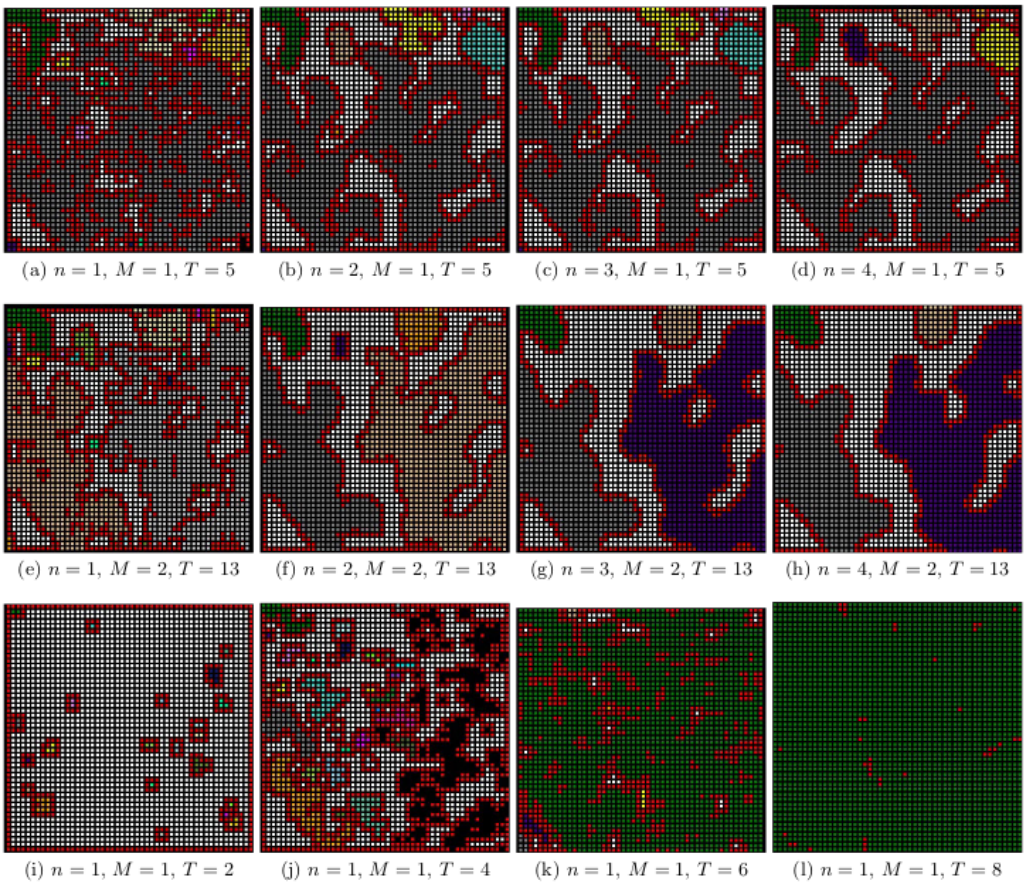
Obliczanie wartości pojedynczej komórki k w dowolnej iteracji automatu komórkowego polega na zastosowaniu po kolei każdego z elementów zbioru C do sąsiadów

z odpowiadających im stopni sąsiedztwa. Dla każdego z takich stopni obliczany jest stosunek liczby *skał* do liczby wszystkich komórek z danego poziomu. Następnie wartość ta jest porównywana z wartością liczbową z odpowiedniego elementu z C za pomocą operatora porównania również tam zawartego. Finalny wynik jest alternatywą kolejnych takich porównań i określa czy komórka k jest *skałą*, czy nie.

Choć Johnson i wsp. sugerują użycie pojedynczej wartości granicznej T [12] zamiast zbioru C takich wartości, to właśnie drugie rozwiązanie pomaga uniknąć niespójności generowanych obszarów. Złożoność problemu pomyślnego wygenerowania jaskini, tzn. spełniającego wszystkie wymienione cechy dobrego labiryntu, rośnie wraz z rozpatrywanym rozmiarem sąsiedztwa, ponieważ wiąże się to bezpośrednio z powiększaniem się zbioru zasad.

4.1.2. Ewoluowanie zbioru zasad automatu komórkowego

Ewolucji może ulegać niemal każdy parametr automatu komórkowego. Johnson i wsp. przedstawili wyniki eksperymentów (rys. 4.1), prezentujące wpływ wybranych parametrów na charakter generowanych obszarów.



Rysunek 4.1: Powyżej przedstawiono wyniki eksperymentów przeprowadzonych z różnymi parametrami automatu komórkowego [12].

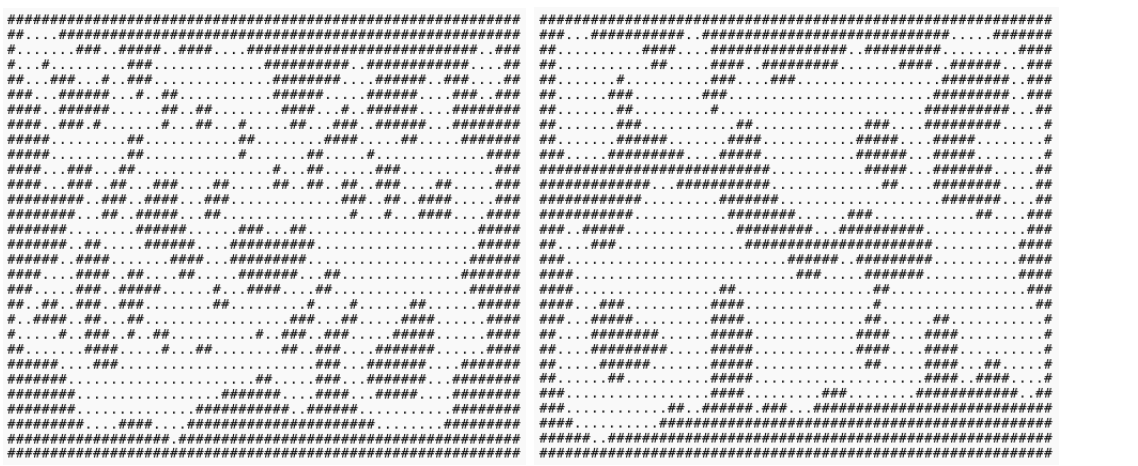
Można łatwo zauważyć, że wraz ze zmianą parametru T rezultaty różnią się od siebie stosunkiem liczby *skał* do *podłóg*. Dla $T = 2$, na wygenerowanym *kafelku* przeważają komórki *skały*, a odwrotny efekt ma ustawienie $T = 8$, gdzie dominują *podłogi*. Dostrajanie automatu za pomocą zwiększania parametru M zdaje się ograniczać liczbę klastrów komórek *podłogi*. Z kolei wzrost liczby iteracji, wyrażonej za pomocą zmiennej n, wpływa wyraźnie na wygładzanie poszczególnych grot. Niestety żadne z zaprezentowanych przykładów nie gwarantuje spełnienia drugiego warunku pomyślnego wygenerowania mapy, opisanym w 2.

W [5] zaproponowano nieco inne podejście do tematu parametryzacji. Autor zauważył problem rozłączności grup i spróbował ten problem rozwiązać samym automatem komórkowym. Otóż przy stałych wartościach $r = 0.45$, $M = 2$ oraz $n = 5$, zmianie uległa forma parametru T, który stał się bardziej złożoną formułą. Nowa zasada determinująca wartość komórki rozważa oba stopnie sąsiedztwa osobno. Reguła ta wyrażona jest wzorem:

$$W'(p) = R_1(p) >= 5 || R_2(p) <= 2$$

gdzie $W'(p)$ to wartość komórki p, a $R_k(p)$ to liczba *skał* w stopniu sąsiedztwa k względem komórki p. Rezultaty zastosowania tej formuły zaprezentowane są na rys. 4.2.

Autor wskazuje na nowy problem, który dotyczy pozostawionych *"filarów"* – pojedynczych komórek *skał* i jednocześnie proponuje jego rozwiązanie przez wykonanie czterech zamiast pięciu iteracji, opisaną wyżej metodą, a następnie dodatkowych trzech dla wygładzenia wyniku, lecz tym razem przy użyciu formuły standardowej, tj. $W'(p) = R_1(p) >= 5$. Dzięki temu usprawnieniu osiągnięto efekt zaprezentowany na rys. 4.2.



Rysunek 4.2: Po lewej mapa posiadająca *filary*. Po prawej grafika przedstawiająca efekt wprowadzenia usprawnienia w postaci dodatkowych iteracji wygładzających [5].

Implementowana wersja automatu komórkowego korzysta z koncepcji zbioru zasad, inspirowanego rozwiązaniem zaproponowanym w [5]. Zbiór ten jest wyznaczony na drodze ewolucji przy zastosowaniu algorytmu genetycznego, do którego regulacji służą następujące parametry:

- POPULATION_SIZE (P_s) – rozmiar populacji, wyrażony za pomocą liczby naturalnej;
- DROP_RATIO (R_d) – procent eliminowanych osobników, wyrażony za pomocą liczby rzeczywistej z zakresu $[0, 1]$;
- MINIMAL_SCORE (S_{min}) – minimalny wynik, jaki genom musi osiągnąć, by z powodzeniem przejść selekcję, przedstawiony w formie liczby naturalnej;
- MUTATION_RATE(R_m) – szansa na wystąpienie mutacji w pojedynczym genie, prezentowany jako liczba rzeczywista z zakresu $[0, 1]$;
- NO_OF_MAPS (n_M) – liczba generowanych i ocenianych map dla każdego z osobników populacji, wyrażony jako dowolna liczba naturalna;
- NO_OF_ITERATIONS (n_I) – liczba iteracji wykonanych przez GA.

Inicjalizowanie populacji polega na stworzeniu listy P_s genomów, z których każdy składa się z unikalnego id, aktualnej punktacji danego genomu (początkowo wynoszącej 0) oraz M-elementowego zbioru zasad. Każdy jego element jest parą, której pierwszy składnik jest losowo wybranym znakiem porównania spośród następującego zbioru $\{<, \leq, >, \geq\}$, drugi to losowa wartość graniczna z przedziału $[0, 1]$.

Każda z n_I-iteracji algorytmu genetycznego przebiega w następujący sposób:

1. Dla każdego elementu obliczana jest jego nowa punktacja. Polega to na wygenerowaniu n_M fenotypów i zastosowaniu na nich funkcji ewaluacyjnych 4.1.2., a następnie zsumowaniu wyników do pojedynczej liczby całkowitej.
2. Ocenione osobniki zostają poddane ewolucji, gdzie:
   (a) Populacja jest sortowana względem wyników z punktu 1.
   (b) Odrzucane jest R_d osobników lub wszystkie, które swoją punktacją nie osiągnęły minimalnego wyniku S_{min}. (minimalna liczba pozostawionych osobników wynosi 2).
   (c) Populacja uzupełniana jest potomstwem, które powstaje przez crossover dwóch losowo wybranych genomów populacji.
   (d) Podczas crossover'u dochodzi również do zmutowania każdego genu z prawdopodobieństwem R_m. Mutacja polega na ponownym, podobnym jak w przypadku inicjalizacji populacji, wylosowaniu operatora oraz liczby granicznej.

Funkcje ewaluacyjne

Do oceniania fenotypu zastosowane zostały trzy bezpośrednie (3.2.3.) funkcje ewaluacyjne w formie kryteriów. Każdy z nich przyjmuje wygenerowany obiekt mapy jako dane wejścia i zwraca liczbę całkowitą.

- Kryterium balansu – podstawowe kryterium, spełnione, gdy stosunek komórek *podłogi* do liczby wszystkich komórek zawiera się w przedziale 20–80%.
- Kryterium ucieczki – warunek, który zrealizowany jest w przypadku gdy liczba podobszarów *podłóg* danego *kafelka* jest większa od 0, a każdy z nich posiada co najmniej dwa wyjścia do dwóch różnych *kafelków*.
- Kryterium dystansu – wymaganie, spełnione, gdy największa z najdłuższych odległości w linii prostej między dwoma komórkami wynosi co najmniej 2 oraz gdy najmniejsza z takich odległości jest większa od 80% MAP_SIZE.

Dla każdego z osobników populacji obliczana jest suma rezultatów aplikacji każdego z powyższych kryteriów do n_M różnych map wygenerowanych z pomocą danego genotypu.

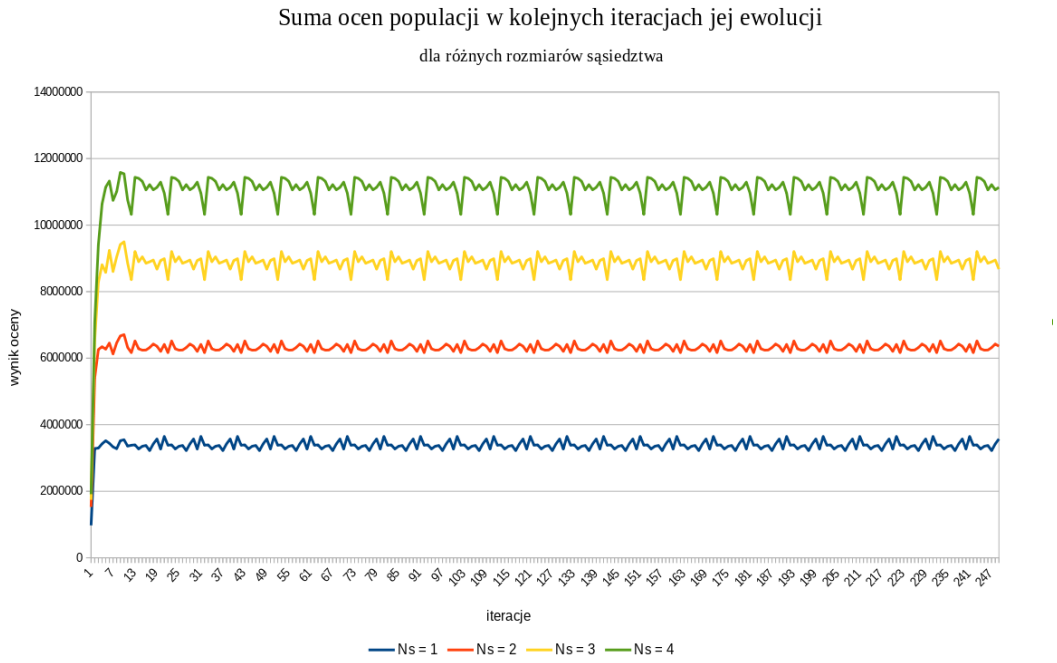
4.2. Wyniki

W ramach niniejszej pracy przeprowadzone zostały liczne testy i próbne ewolucje oraz cztery eksperymenty właściwe. W tym rozdziale zostaną przedstawione wyniki prac wraz z opisami.

4.2.1. Funkcje oceny a postęp populacji podczas ewolucji

Wspomniane wcześniej cztery badania polegały na każdorazowym wykonaniu pięciuset iteracji algorytmu ewolucyjnego dla różnych wartości parametru M. Rozmiar *kafelka* od pewnej wartości nie wpływa na jego charakterystykę, dlatego w badaniu przyjęto $MAP\_SIZE = 50$. Pozostałe parametry zostały ustalone na podstawie prac [5] i [12] oraz testów wstępnych i wynoszą odpowiednio $R_m = 0.4$, $n_I = 8$, $n_M = 20$, $R_d = 0.6$ i $P_s = 200$.

Wykres przedstawiony na rys. 4.3 obrazuje postęp sumy wartości funkcji ocen populacji na przestrzeni 250 z 500 iteracji. Pozostałe 250 powtórzeń algorytmu ewolucyjnego charakteryzowało się identycznymi cechami, dlatego ich prezentowanie nie wniosłoby dodatkowej wartości, a jedynie gorszą czytelność.
Dla każdej z wartości M obserwujemy podobną tendencję- niski wynik podczas kilku pierwszych iteracji, który szybko postępuje aż do osiągnięcia wartości, wokół której następnie nieznacznie oscyluje. Diagram pokazuje również, że im większy rozmiar



Rysunek 4.3: Wykres przedstawia sumę wyników ocen populacji o różnych wartościach M dla dwustu pięćdziesięciu iteracji.

sąsiedztwa M, tym niższa maksymalna osiągnięta suma ocen populacji. Zwiększając M, zwiększa się zbiór możliwych wartości parametru C, przez co trudniej wyewoluować bardziej obiecujące osobniki.

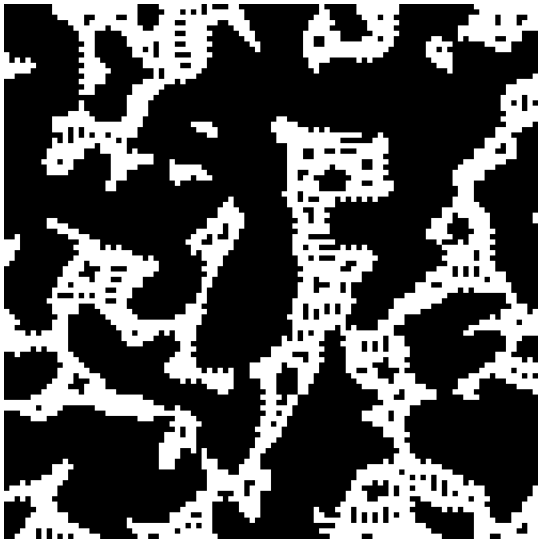
4.2.2. Wpływ parametrów CA na strukturę generowanych map

Kafelki budowane przez algorytm komórkowy ściśle zależą od wartości parametrów. Poniżej przedstawione są przykładowe rezultaty dla różnych wartości zbioru warunków C, które zostały wyewoluowane jeszcze w trakcie fazy testowania i badania pojęcia *ciekawego labiryntu*.

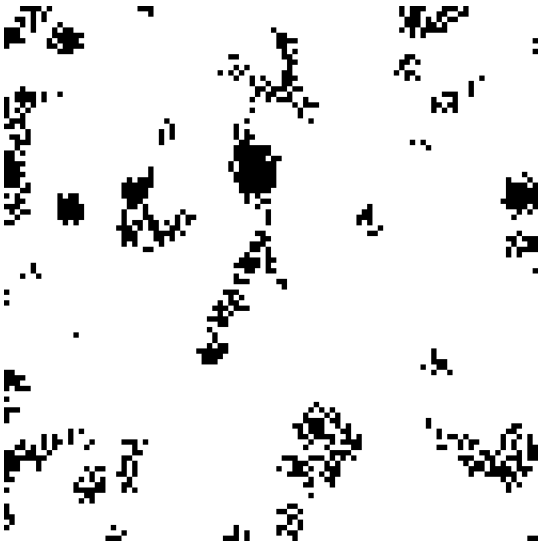
Rys. 4.4 pokazuje problem małych rozłącznych grup komórek *podłogi*. Utworzone korytarze są nietrywialne i pełne małych przeszkód- filarów, ścian itp. Lista zasad C odpowiada regułom zaproponowanym w [5].

Genotyp, który posłużył do wygenerowania *kafelka* zaprezentowanego na rys. 4.5 jest bardzo ciekawym przykładem, w którym większość obszaru jest pusta. Nie jest to jednak koniecznie niechciany efekt. Można sobie łatwo wyobrazić przypadek, kiedy chcielibyśmy utworzyć fragment labiryntu, który reprezentowałby otwartą przestrzeń.

Na rys. 4.6 przedstawiona jest mapa, cechująca się pozornym brakiem spójności i struktury utożsamianej ze stereotypowym labiryntem, czy jaskinią. Nie mniej ponownie taki efekt może być pożądany, w przypadku gdy fragment wygenerowanego



Rysunek 4.4: $C = [['>', 1], ['>=', 0.5], ['<', 0.125]], M = 2$



Rysunek 4.5: $C = [['<=', 0.136], ['<', 0.438], ['>=', 0.991], ['>', 0.800]], M = 3$

terenu ma reprezentować przestrzeń trudną do przebycia, jak np. las stalagmitów, stalagnatów, czy innych skalnych formacji.

Rys. 4.7 prezentuje jedne z pierwszych eksperymentów dla $M = 3$, na których z powodzeniem można zaobserwować strukturę nietrywialnego, złożonego labiryntu. Jednak wciąż problemem pozostaje brak spójności wewnątrz *kafelka*, czyli istnienie niedostępnych zbiorów komórek *podłogi*.

Grafiki zaprezentowane na rys. 4.8-4.11 przedstawiają *kafelki* wygenerowane za pomocą osobników populacji z eksperymentów opisanych w 4.2.1.

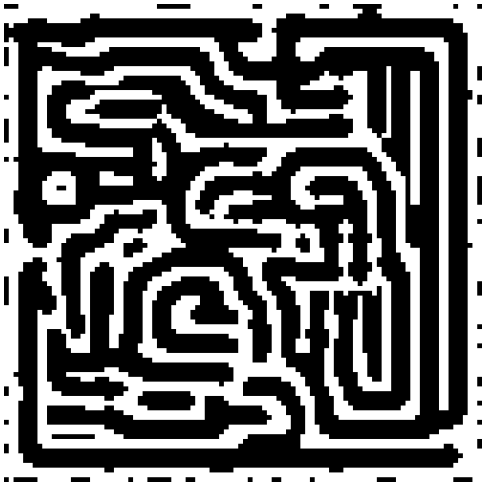
Najczęściej występującym wzorem labiryntu dla populacji poddanej ewolucji z $M = 1$ był rys. 4.8a (tzn. najwięcej różnych genotypów skutkowało tym samym, przedstawionym fenotypem). Z kolei kafelek z rys. 4.8c, posiadał jednostkowe wystą-

Rysunek 4.6: $C = [['>=', 0.838], ['<', 0.035], ['>=', 0.800], ['>', 0.869], ['<=', 0.299]]$, $M = 4$

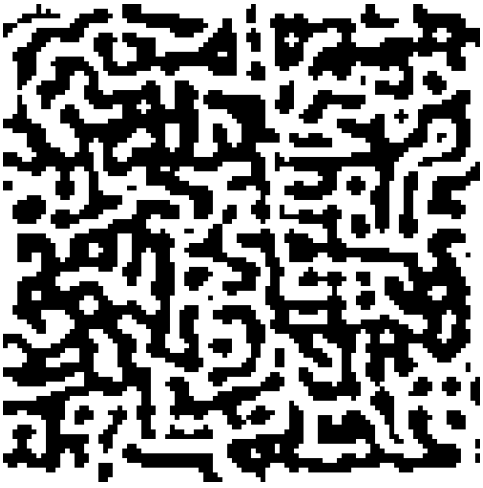
pienie w obrębie całej populacji. Warto zwrócić uwagę na różnicę między rys. 4.8b a 4.8c, gdzie oba C mają jednakowe pierwsze składniki i różnią się dopiero drugim elementem zbioru. Pomimo takiego podobieństwa genotypów, fenotypy posiadają zupełnie odmienne cechy. Natomiast rys. 4.8b i rys. 4.8d pomimo tego, że posiadają skrajnie różne drugie elementy zbioru C (tj. odmienny znak i inną wartość), to wykazują wiele wspólnych własności. Oba *kafelki* charakteryzują się choćby wąskimi korytarzami, a pewne centralne obszary są wręcz identyczne.

Na rys. 4.9 pokazani są reprezentanci populacji po 500 iteracjach algorytmu ewolucyjnego z $M = 2$. Osobnik z rys. 4.9a tworzy bardzo intuicyjne, wąskie labirynty o korytarzach szerokości zaledwie jednej komórki. Niestety posiada negatywną cechę pozostawiania pojedynczych, odizolowanych komórek *podłogi*. Rys. 4.9b przedstawia fenotyp osobnika, który choć posiada C o mocy 3, to produkuje bardzo podobne mapy do genotypów z $M = 1$ (rys. 4.8a). Pozostali przedstawiciele charakteryzują się nieregularnymi kształtami korytarzy o różnych szerokościach oraz podobnie wyglądającymi obszarami w pobliżu narożników *kafelka*. Dodatkowo rys. 4.9c i 4.9d posiadają tzw. kolumny, czyli odizolowane, mało liczne (1-3) grupy komórek *skały*.

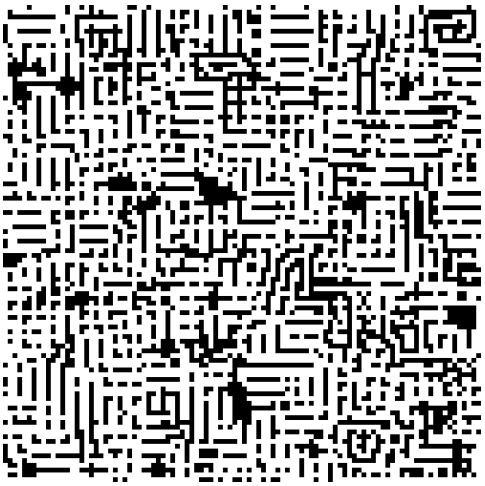
Choć wykres dla $M = 3$ wskazuje na to, że populacja osiąga trzeci wynik z kolei, to przeprowadzony eksperyment poskutkował populacją, która w rzeczywistości jest najlepszą spośród wszystkich czterech. Rys. 4.10a posiada strukturę charakteryzującą się prostymi, głównie wertykalnymi, przerywanymi liniami oraz licznymi *kolumnami*. Warto zauważyć, że rys. 4.10b, 4.10c i 4.10d nawiązują do rys. 4.9c, przy czym największe podobieństwo zachodzi między rys. 4.10b a 4.9c Wszystkie wymienione cechują się licznymi *kolumnami*, stosunkowo małymi przestrzeniami oraz podobnymi strukturami w pobliżu narożników *kafelka*. Jednak najciekawsze osob-



(a) $C = [['<=', 0.855], ['<=', 0.679], ['<=', 0.169], ['>', 0.892]]$



(b) $C = [['<=', 0.136], ['<', 0.438], ['>=', 0.991], ['>', 0.800]]$



(c) $C = [['<=', 0.588], ['>=', 0.989], ['>', 0.889], ['<', 0.397]]$

Rysunek 4.7: Wyniki pierwszych eksperymentalnych ewolucji dla $M = 3$

niki zostały zaprezentowane na rys. 4.10e oraz 4.10f Ten pierwszy również cechuje się licznymi kolumnami, prostymi liniami, ale co najbardziej interesujące- tworzy relatywnie długie i wyraźne pętle i nie wykazuje tendencji do tworzenia *nieużytków*. Drugi przypadek wygląda również obiecująco- złożone systemy korytarzy przez niego tworzonych, są nierówne, nieco chaotyczne, posiadają liczne *ślepe zaułki*, a czasem przeobrażają się w obszerne pomieszczenia. Niestety występują w nim nieosiągalne, odosobnione grupy komórek *podłogi – nieużytki*.

Ewolucja populacji dla $M = 4$ przyniosła fenotypy (rys. 4.11), które raczej należały do mniejszości w poprzednich eksperymentach. Przeważająca większość osobników przejawiała tendencję do tworzenia labiryntów o korytarzach prostych i wąskich, w dodatku z wieloma nieużytkami, tudzież nagle urywającymi się korytarzami. Obrazują to grafiki 4.11a, 4.11b i 4.11c, przy czym tylko ostatni z fenotypów,

(a) $C = [['<', 0.310], ['<', 0.190]]$ (b) $C = [['<=', 0.656], ['>', 0.654]]$

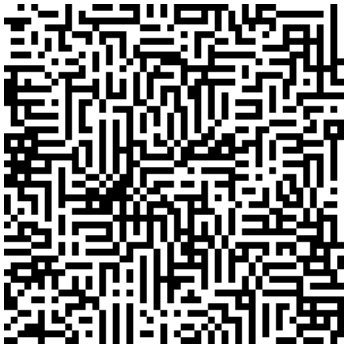
(c) $C = [['<=', 0.656], ['<', 0.306]]$ (d) $C = [['>', 0.001], ['<=', 0.177]]$

Rysunek 4.8: Najczęściej występujące wzory dla $M = 1$.

pomimo, że nie wolny od *nieużytków*, prezentuje się jako możliwy do użycia. Ostatnie trzy wizualizacje przedstawiają fenotypy obfitujące w *kolumny* oraz cechujące się dość chaotycznym układem korytarzy, ale pomimo tego nie posiadają większych wad. Na rys. 4.11f można dostrzec liczne podobieństwa do fenotypów z poprzednich eksperymentów (zaprezentowane na rys. 4.8d, 4.9c oraz 4.10b).

4.2.3. Zachowanie łączności generowanych *kafelków*

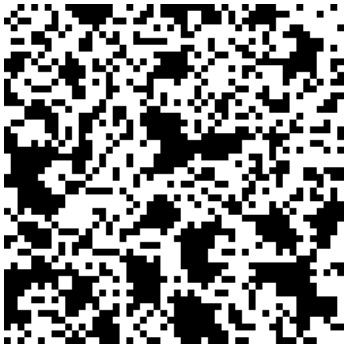
Z opisanych w 4.2.1. i przedstawionych na rys. 4.8-4.11 wyników eksperymentów, zostało wybranych dziesięciu najciekawszych osobników. Każdy z nich został zaprezentowany na rys. 4.12-4.21 w konfiguracji dziewięciu sąsiadujących ze sobą *kafelków*, w celu pokazania stopnia zachowania własności *sklejalności* generowanych *kafelków*. Do prezentacji użyto różnych parametrów C i odpowiadającym im M,



(a) $C = [[' <=', 0.0154], [' >', 0.784], [' <', 0.365]]$

(b) $C = [[' <=', 0.595], [' <=', 0.190], [' <', 0.110]]$

(c) $C = [[' >=', 0.131], [' >=', 0.916], [' <=', 0.037]]$



(d) $C = [[' >', 0.0935], [' >=', 0.660], [' >', 0.914]]$

(e) $C = [[' <', 0.137], [' >', 0.843], [' >', 0.804]]$

Rysunek 4.9: Przykładowe *kafelki* dla $M = 2$.

w zależności od liczebności zbioru C. Rozmiar każdego z *kafelków* jest stały i wynosi $MAP\_SIZE = 100$.

Rys. 4.12 pomimo wad omówionych już w 4.2.1. posiada pewne cechy, które można dostrzec dopiero po wygenerowaniu *kafelka* startowego wraz z jego sąsiadami. Na grafice wyraźnie widać granice kolejnych *kafelków*, co ponownie może być niepożądanym efektem, ale nie musi. Podobne granice widać jeszcze na rys. 4.13, lecz tu już są dużo mniej wyraźne. Problemu nie odnotowano w pozostałych przypadkach. Najlepszym przykładem ukrycia granicy *kafelka* jest zaprezentowany na rys. 4.15. Choć rozmiar prezentowanych grafik nie pozwala na jednoznaczne określenie szczegółowych cech, to przy uważnej analizie widać, że dalsze przykłady nie posiadają wad w postaci zamkniętych grup granicznych, wspomnianych wyraźnych granic, itp.

4.3. Wnioski

Największym wciąż nierozwiązanym problemem pozostają odizolowane pomieszczenia wewnątrz generowanych *kafelków*. Pierwsze rozwiązanie tego problemu to łą-

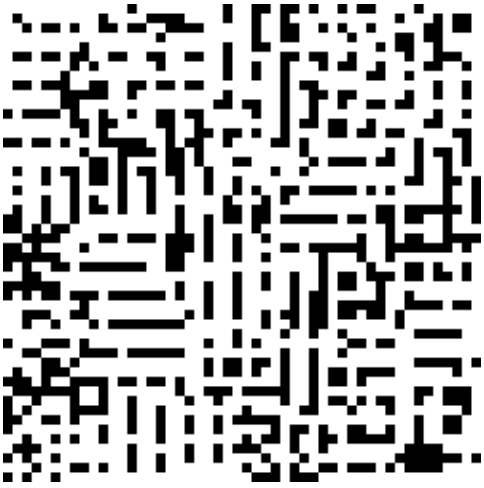
czenie tych pomieszczeń korytarzami. To jednak rodzi tylko więcej problemów, jak np.: z jaką inną podgrupą należy wykonać połączenie; jaki ma być korytarz – szeroki czy wąski, prosty czy kręty, o stałej szerokości czy zmiennej; z jakiego miejsca – z takiego o najmniejszej odległości do podgrupy sąsiedniej czy z takiego który zapobiegnie przecinaniu się korytarzy (tu znowu, *sztuczne* korytarze mogą się przecinać, czy też nie powinny). Ponadto metoda ta, w dowolnym jej wariancie sprawiłaby, że na końcowy wynik sam automat komórkowy miałby tylko częściowy, żeby nie powiedzieć nieznaczny wpływ, przez co sam proces ewolucji parametrów CA byłby niemiarodajny.
Alternatywnym rozwiązaniem jest tzw. zalewanie obszarów nieposiadających ani jednego wyjścia z *kafelka*. Jest to lepsze rozwiązanie i wymaga rozważenia, jednak jest ono ryzykowne i powinno być stosowane tylko w wypadku gdy nieużytki zajmują określony procent całego *kafelka*.
Ostatnią solucją jest zignorowanie problemu i obrócenie go w mechanikę gry, w której jedną z aktywności byłoby przedostawanie się przez ściany.
Choć w [5] autor rozwiązał ww. problem odpowiednio konfigurując automat komórkowy, to z eksperymentów wynika, że jest możliwe otrzymanie pojedynczego *kafelka* bez rozłącznych podgrup, lecz przedstawiona implementacja nie pozwala na zagwarantowanie zachowania tej właściwości dla każdego z nich.

Wykres prezentowany na rys. 4.3 sugeruje, że proces samej ewolucji również wymaga usprawnienia. Przede wszystkim dodatkowa, symulacyjna funkcja ewaluacyjna bazująca na systemie agentów, mogłaby wnieść dużo skuteczniejszą formę eliminacji osobników generujących *nieciekawe* mapy, które są błędnie wysoko oceniane. Efekt wypłaszczenia wykresu dla każdego z eksperymentów może być rezultatem małej różnorodności populacji, lub zbyt łatwego uzyskiwania przez osobniki wyników maksymalnych. Z kolei bardzo szybki wzrost wartości sumy wyników ewaluacji osobników populacji na przestrzeni zaledwie kilku pierwszych iteracji sugeruje błąd po stronie procesu ewaluacji lub selekcji. Na poprawę tego aspektu mogłoby wpłynąć porzucenie rozwiązania stosującego wynik minimalny S_{min} jako próg selekcji, co prawdopodobnie obecnie jest przyczyną małej różnorodności genetycznej wśród populacji.
Kolejnym potencjalnym usprawnieniem, mogłoby być zwiększenie liczby funkcji ewaluacyjnych. Niosłoby to dodatkową korzyść w postaci większej kontroli wyników. Na grafikach rys. 4.8-4.21 przedstawionych zostało wiele wariantów labiryntów możliwych do wygenerowania. Po sprecyzowaniu oczekiwań i zdefiniowaniu kryteriów można by z powodzeniem wprowadzić funkcje ewaluacyjne skupione na mierzeniu wcześniej określonych cech. Gwarantowałyby one jednocześnie bardziej zróżnicowane wyniki poszczególnych osobników oraz bardziej przewidywalne fenotypy.

Tzw. *sklejalność kafelków* jest na dużo lepszym poziomie. Prawdopodobnie przez fakt, że podczas generowania *kafelka* pod uwagę brane są również granice sąsiadów. Kolejnym czynnikiem, który może pozytywnie wpływać na tę cechę jest kryterium ucieczki, gwarantujące każdej podgrupie *podłóg* posiadanie połączenia z brze-

giem *kafelka*.

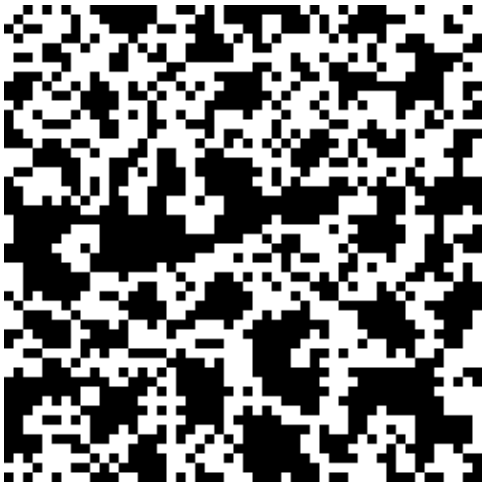
Jednakże i tu można doszukać się wad rozwiązania. Na rys. 4.12 i 4.13 wyraźnie widać korytarze graniczne, które prowadzą do wyjątkowo trywialnych labiryntów, ponieważ można je eksplorować niemal wcale nie zagłębiając się do centrum któregokolwiek z *kafelków*, a jedynie poruszając się po granicy dwóch sąsiednich obszarów. Ten problem prawdopodobnie rozwiązałoby odpowiednie kryterium polegające na zliczaniu punktów kontaktu z sąsiadami.



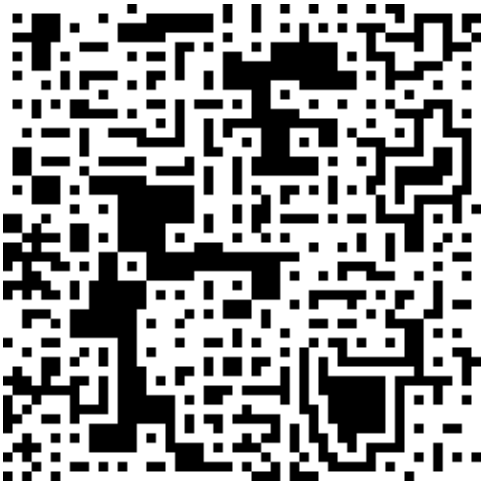
(a) $C = [[' <=', 0.860], [' >', 0.956], [' >', 0.917], [' <', 0.337]]$

(b) $C = [[' >=', 0.186], [' >', 0.956], [' <', 0.106], [' >', 0.777]]$

(c) $C = [[' >', 0.901], [' >', 0.956], [' <', 0.106], [' >', 0.602]]$



(d) $C = [[' >', 0.901], [' >', 0.694], [' <=', 0.184], [' <=', 0.217]]$

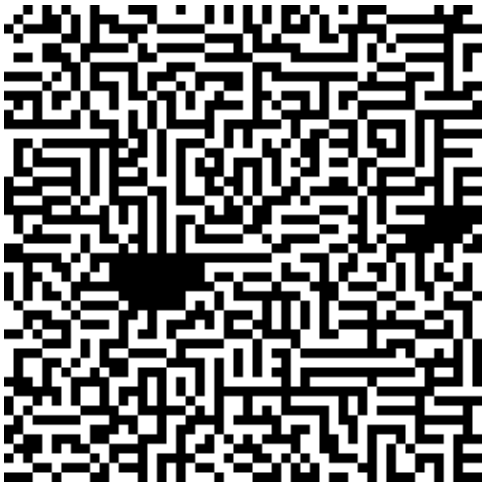


(e) $C = [[' <=', 0.007], [' >', 0.972], [' <=', 0.184], [' <', 0.347]]$

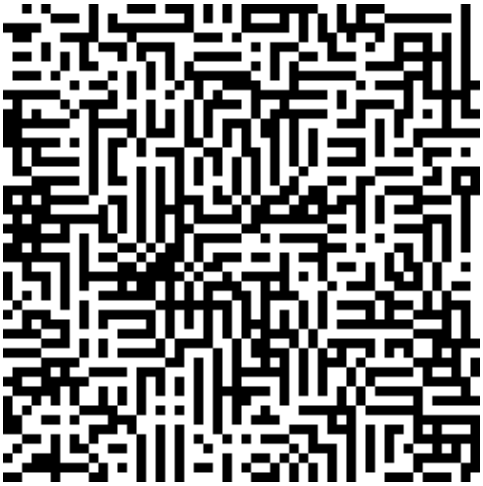
(f) $C = [[' <=', 0.007], [' <=', 0.358], [' >', 0.787], [' <=', 0.089]]$

Rysunek 4.10: Przykładowe *kafelki* dla $M = 3$.

(a) $C = [['<', 0.667], ['>', 0.873], ['<=', 0.302], ['>=', 0.749], ['<=', 0.1956]]$

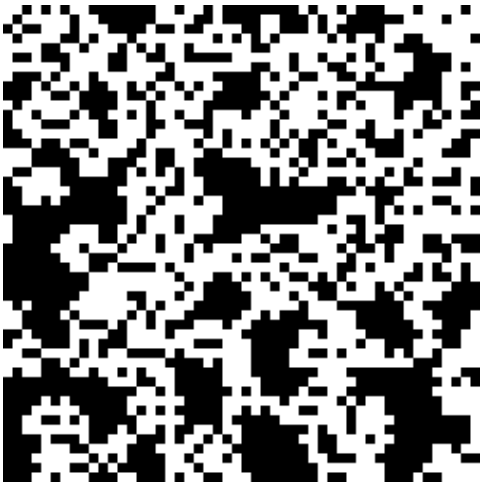


(b) $C = [['<=', 0.80], ['>=', 0.678], ['<', 0.213], ['<=', 0.061], ['<', 0.1210]]$



(c) $C = [['<=', 0.811], ['>', 0.873], ['<', 0.325], ['<', 0.084], ['<', 0.0615]]$

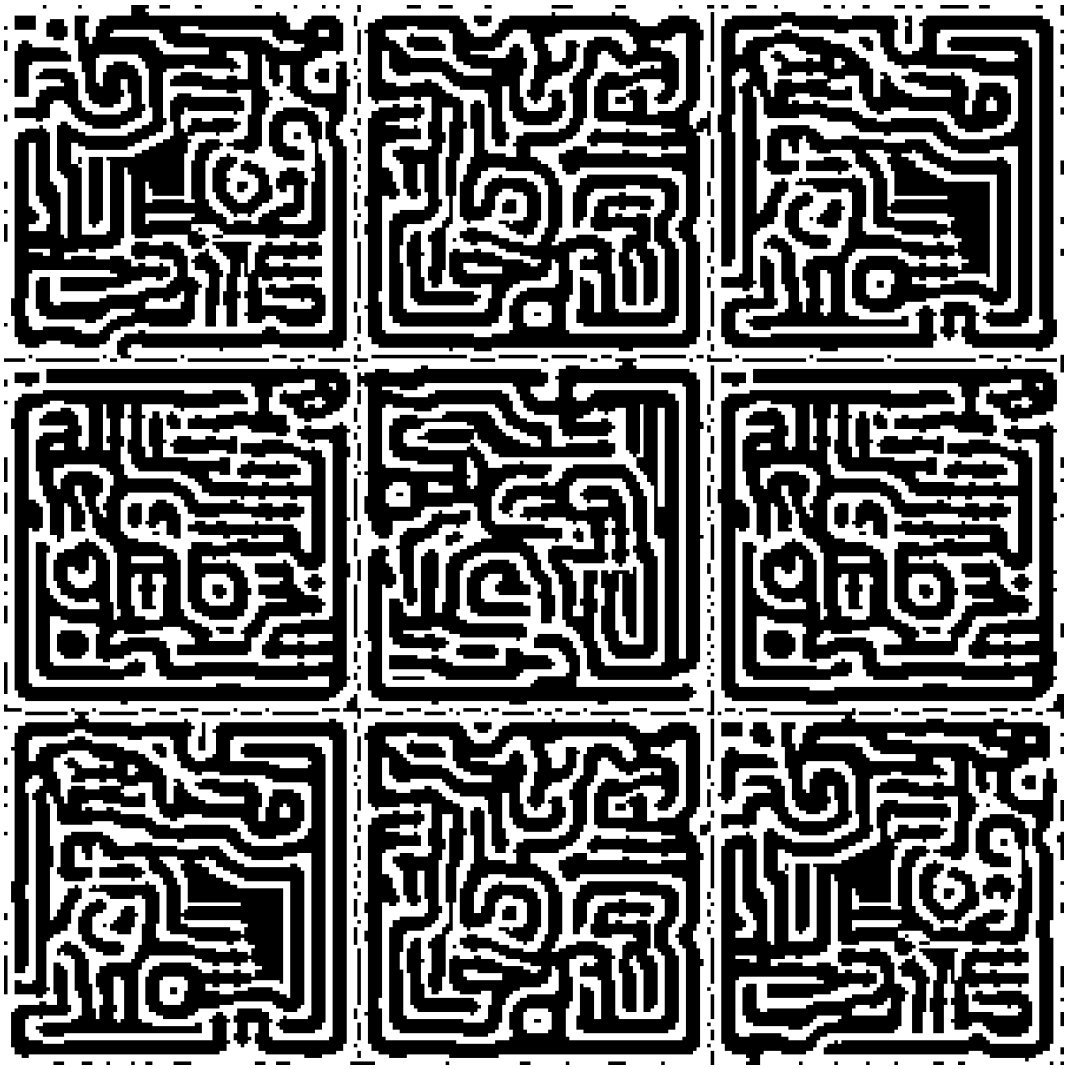
(d) $C = [['>', 0.919], ['>', 0.849], ['>', 0.630], ['<=', 0.242], ['>=', 0.7228]]$



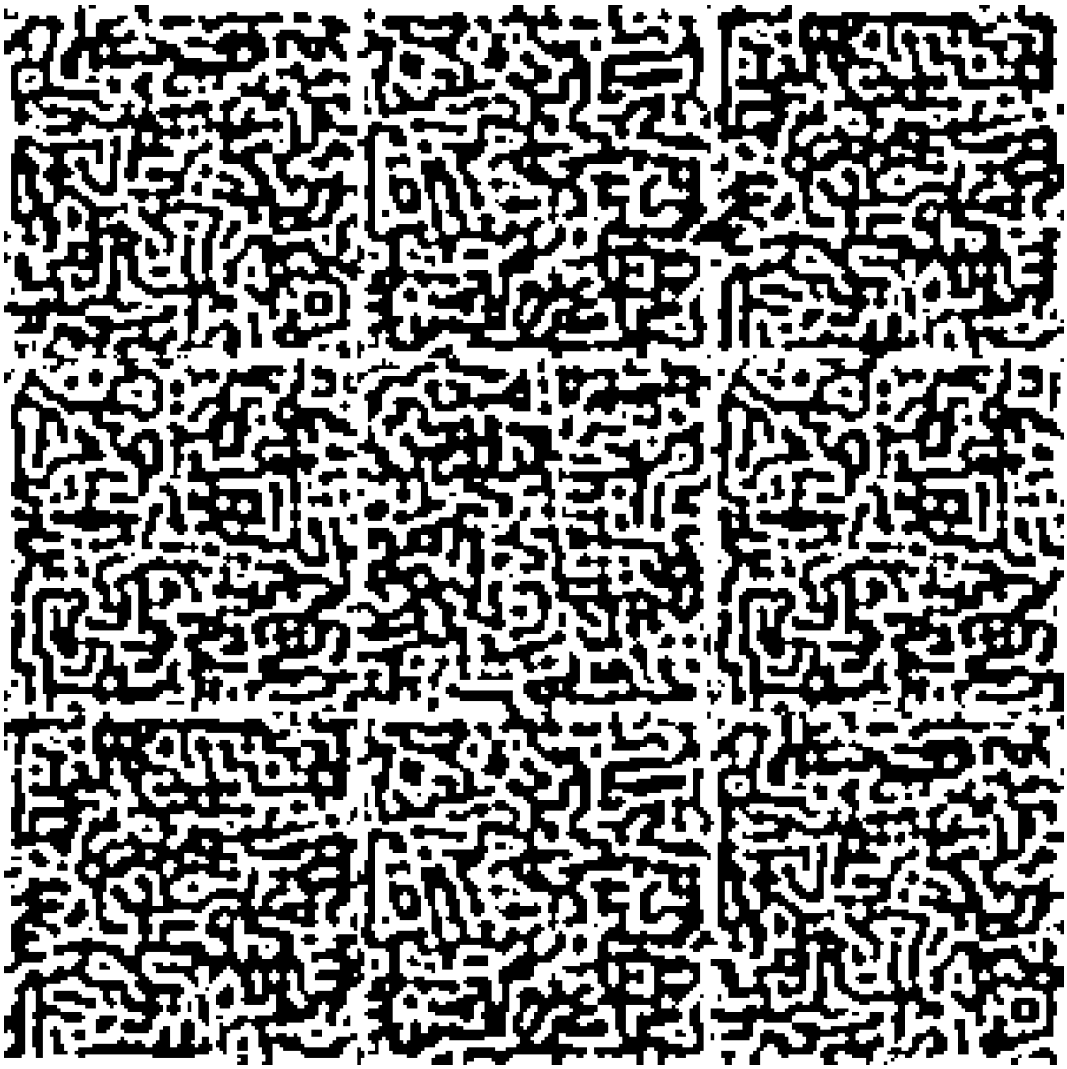
(e) $C = [['>', 0.919], ['>', 0.625], ['>=', 0.800], ['<=', 0.061], ['>=', 0.8792]]$

(f) $C = [['>=', 0.941], ['>', 0.941], ['>=', 0.800], ['<=', 0.061], ['>=', 0.8285]]$

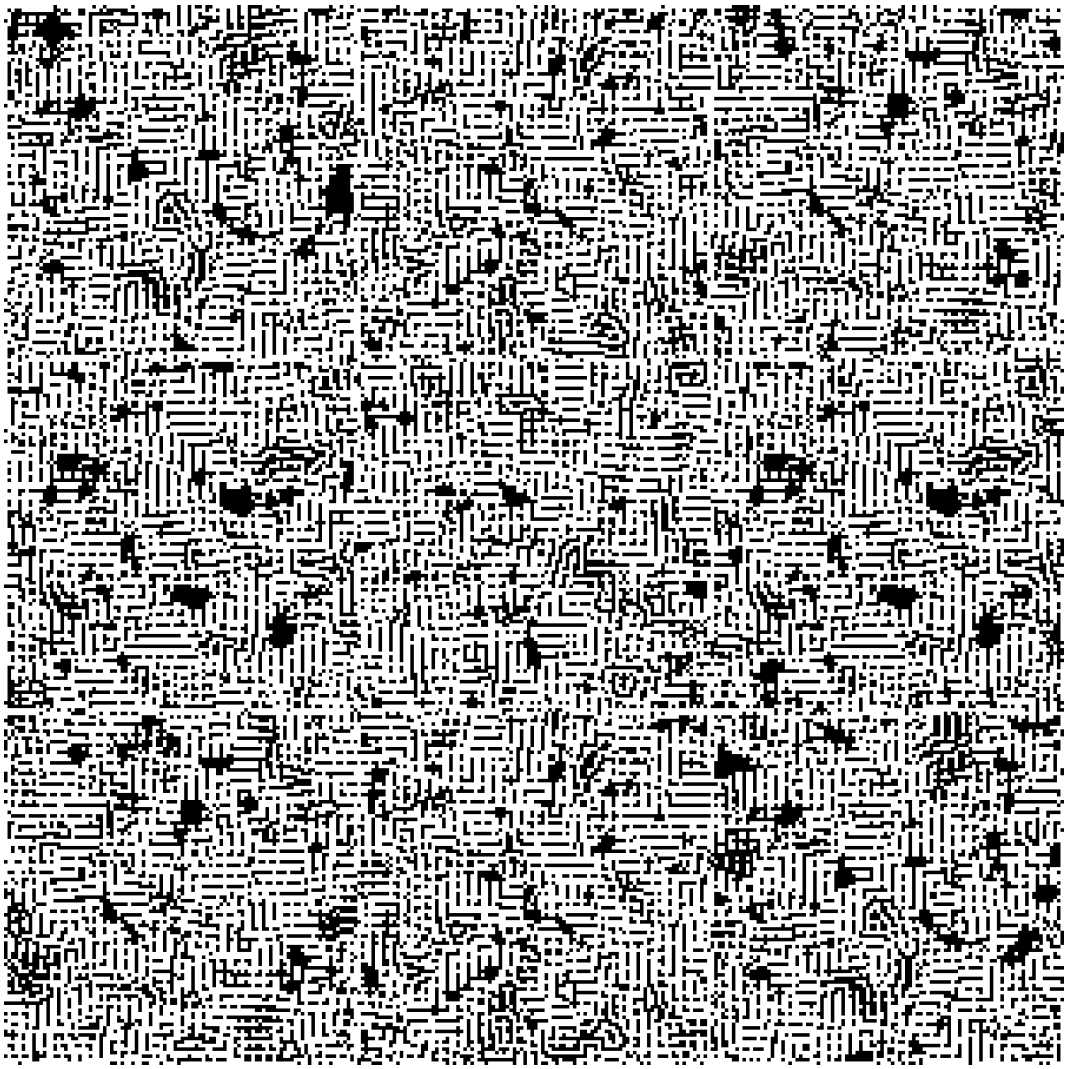
Rysunek 4.11: Przykładowe *kafelki* dla $M = 4$.



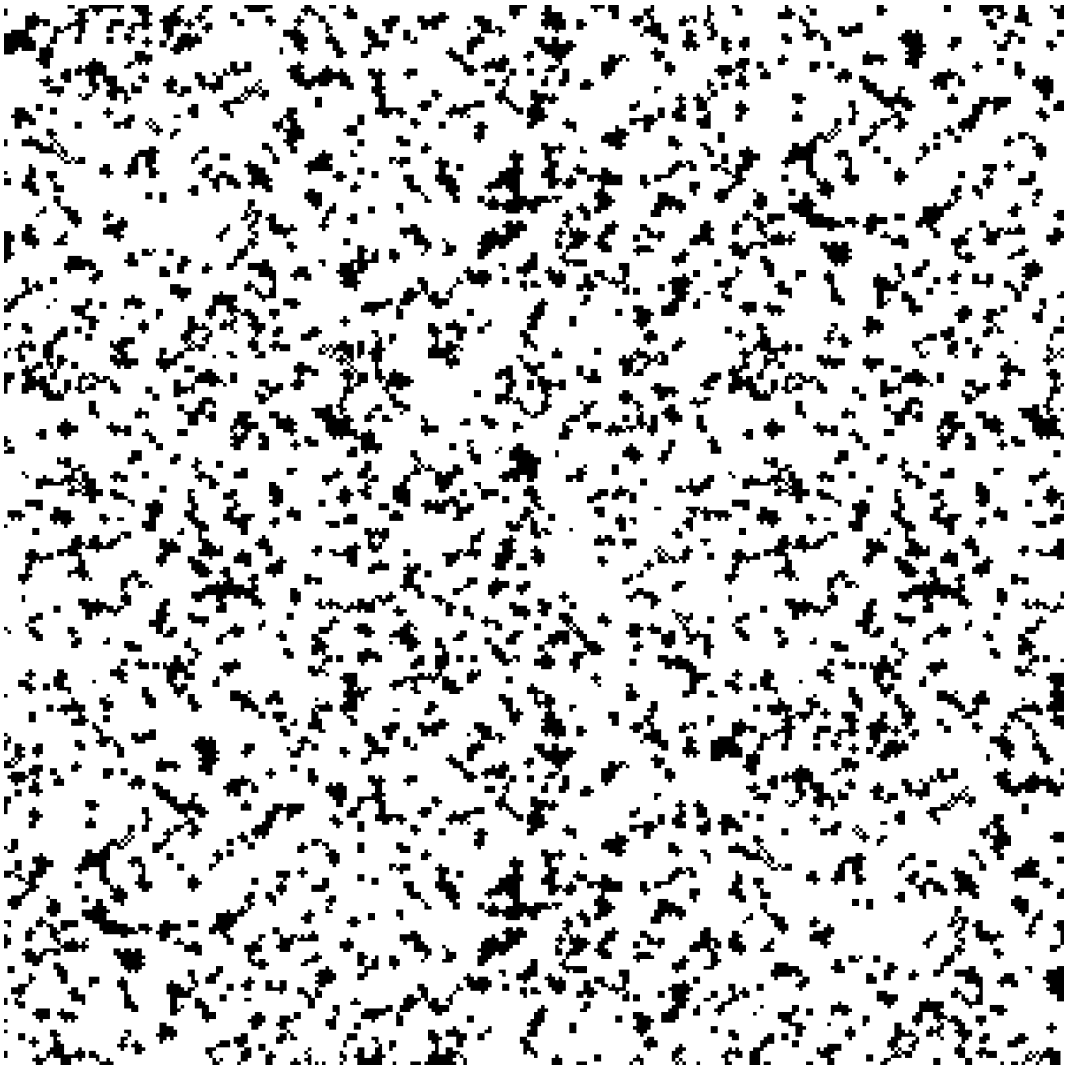
Rysunek 4.12: $C = [['<=', 0.855], ['<=', 0.679], ['<=', 0.169], ['>', 0.892]]$



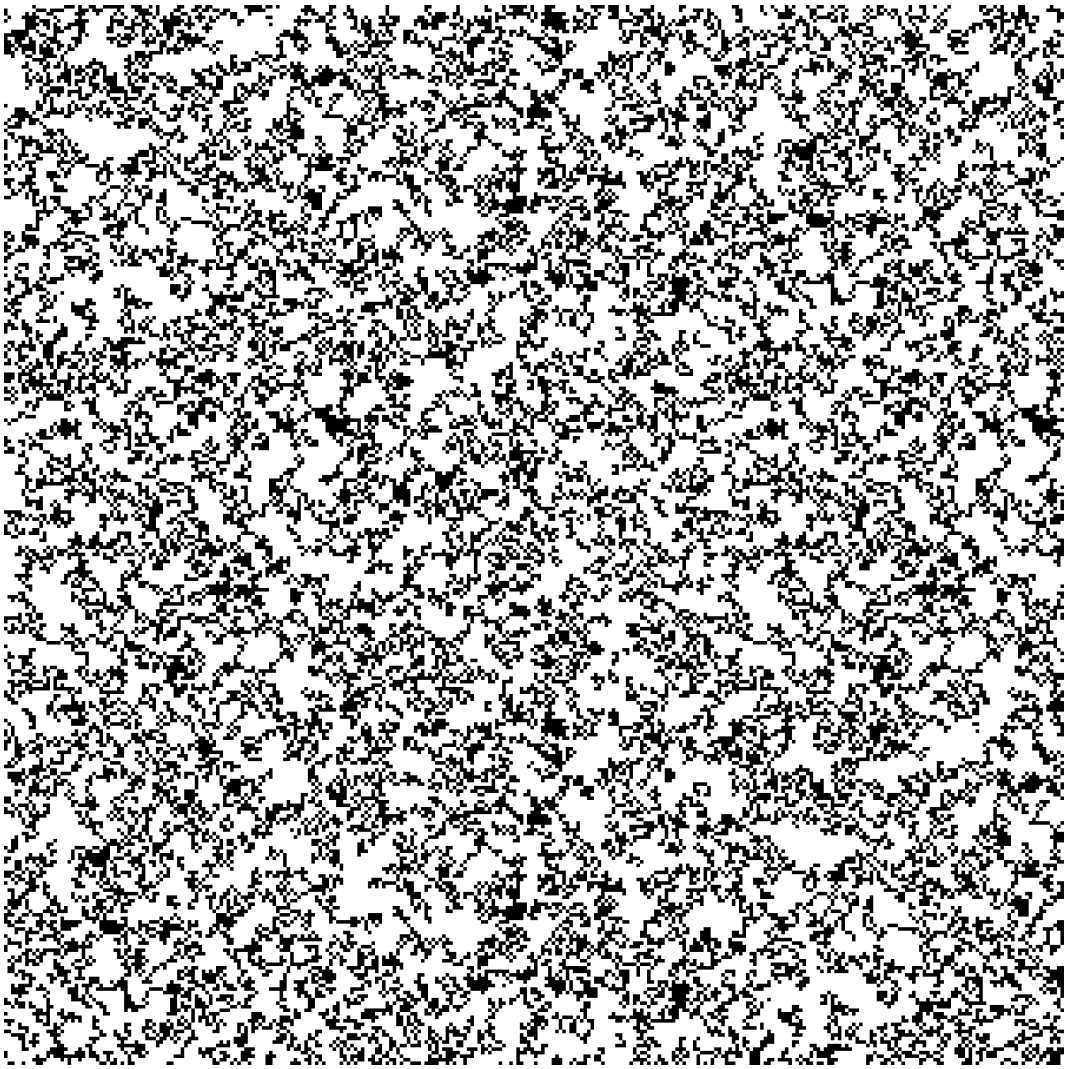
Rysunek 4.13: $C = [['<=', 0.136], ['<', 0.438], ['>=', 0.991], ['>', 0.800]]$



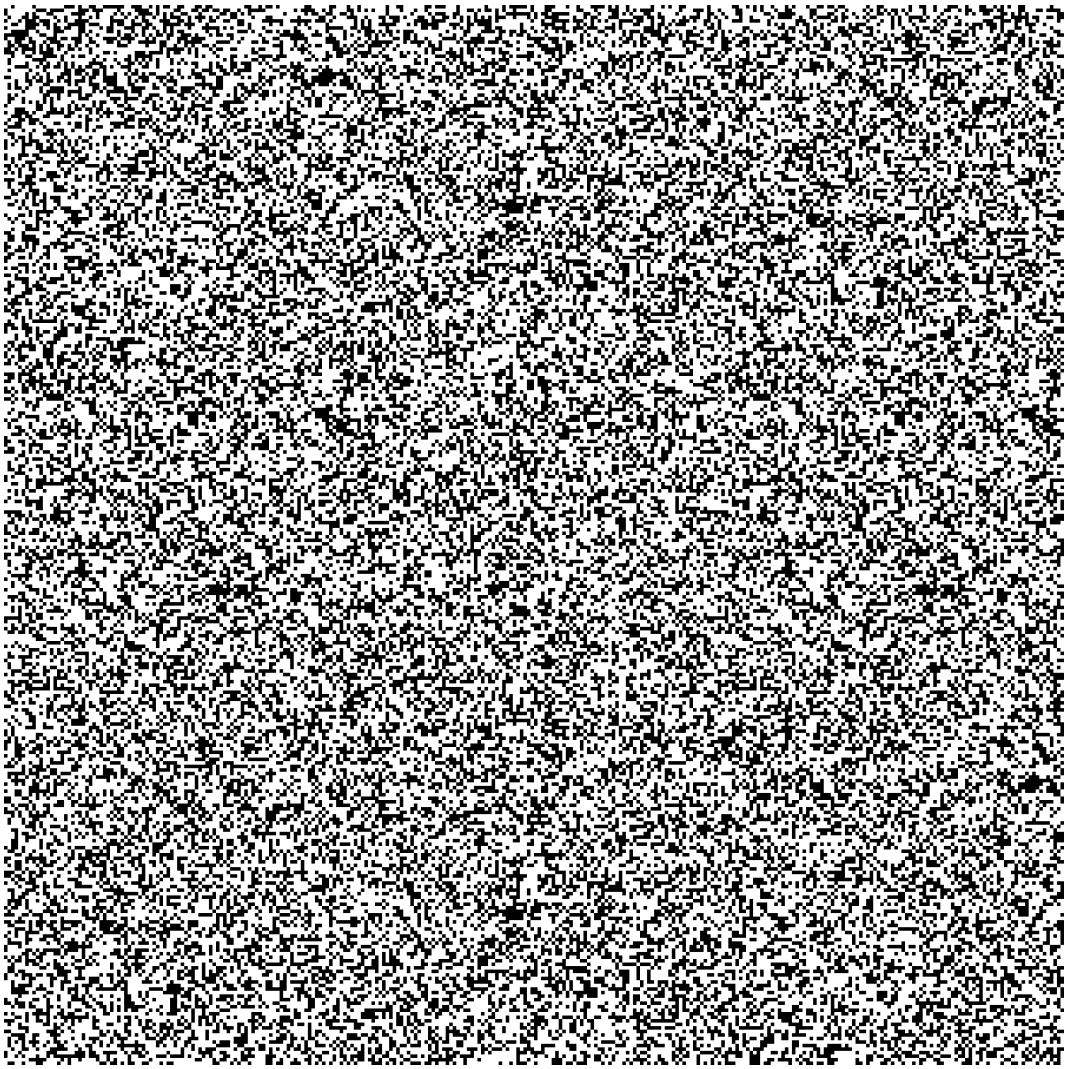
Rysunek 4.14: $C = [['<=', 0.588], ['>=', 0.989], ['>', 0.889], ['<', 0.397]]$



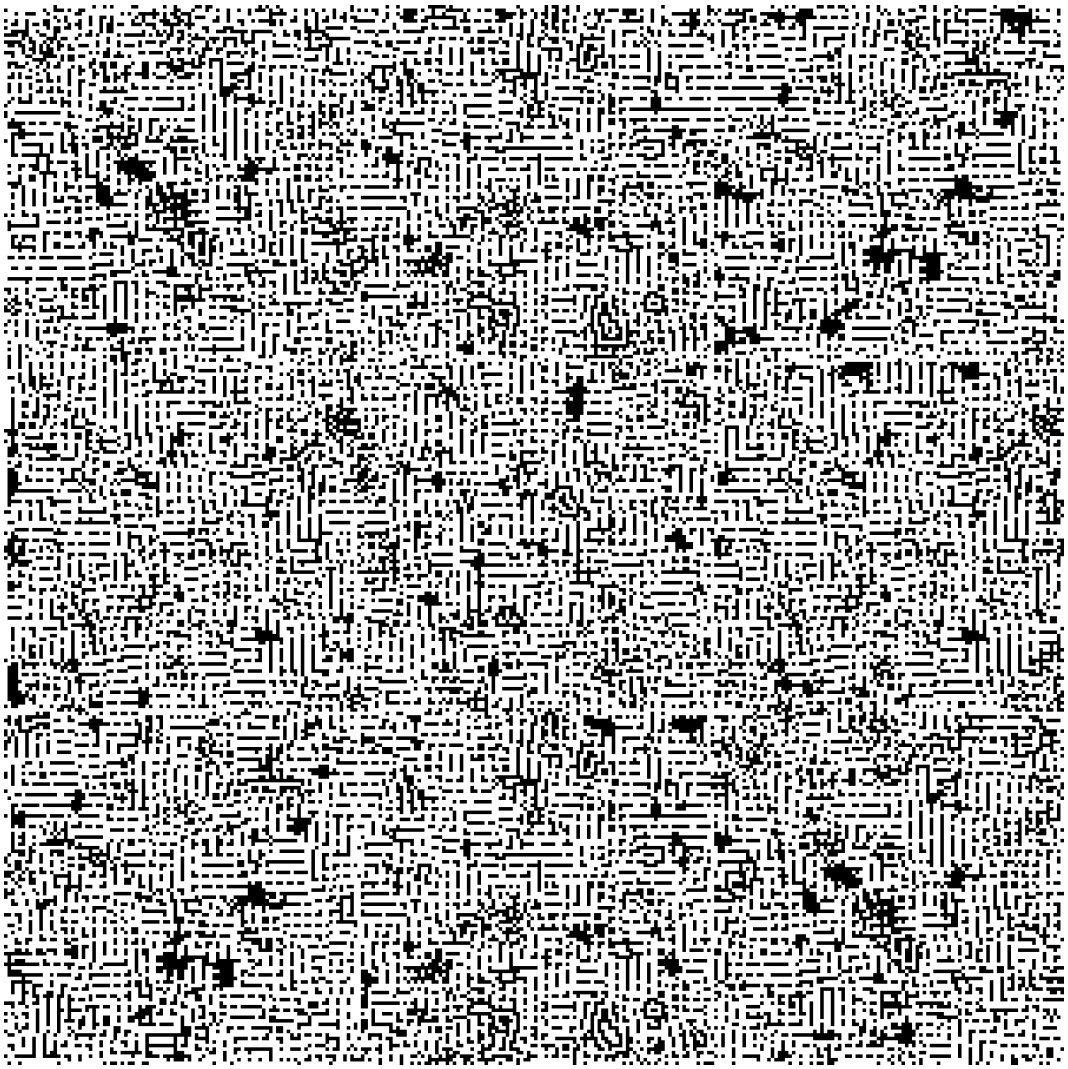
Rysunek 4.15: $C = [['<=', 0.656], ['<', 0.306]]$



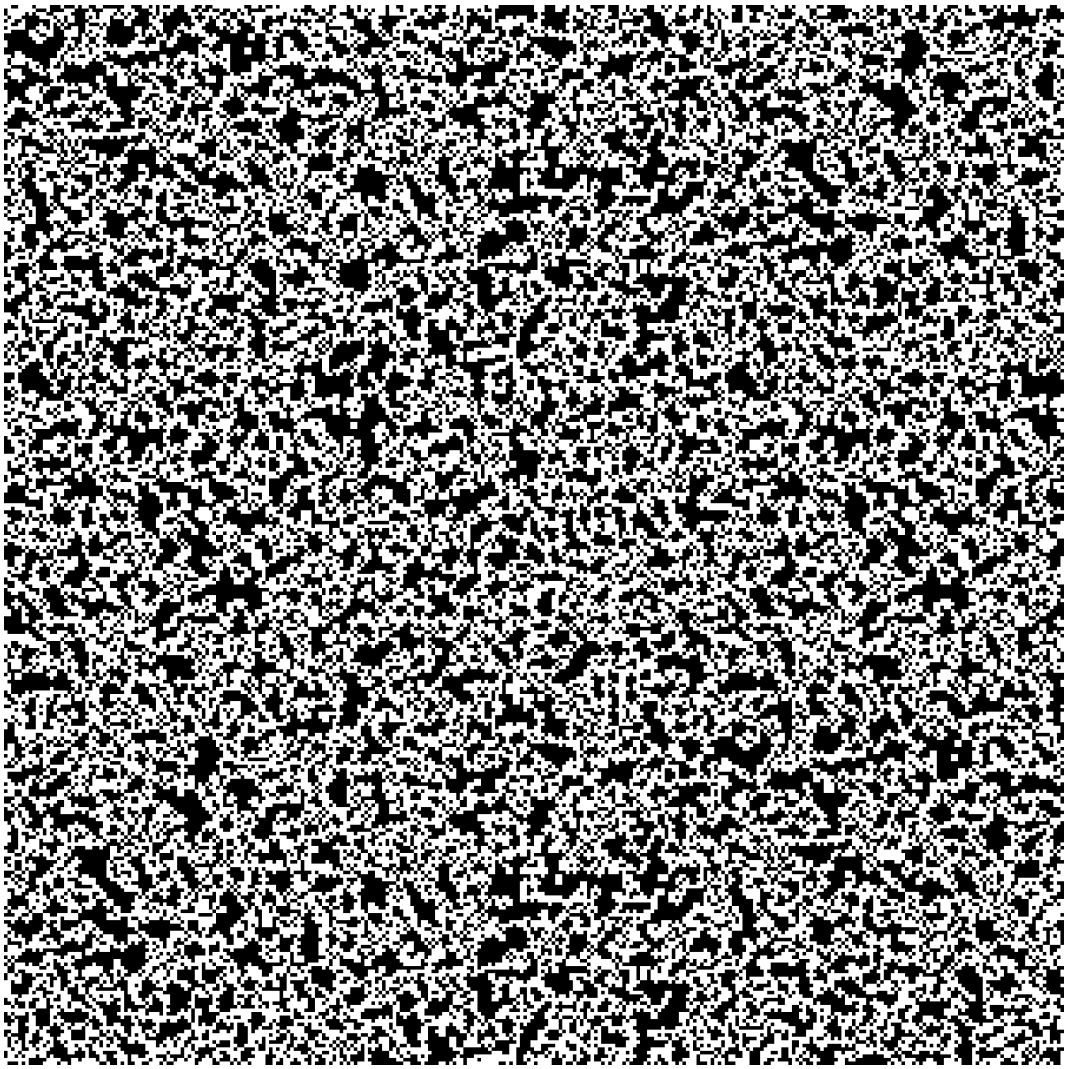
Rysunek 4.16: $C = [['<=', 0.595], ['<=', 0.190], ['<', 0.110]]$



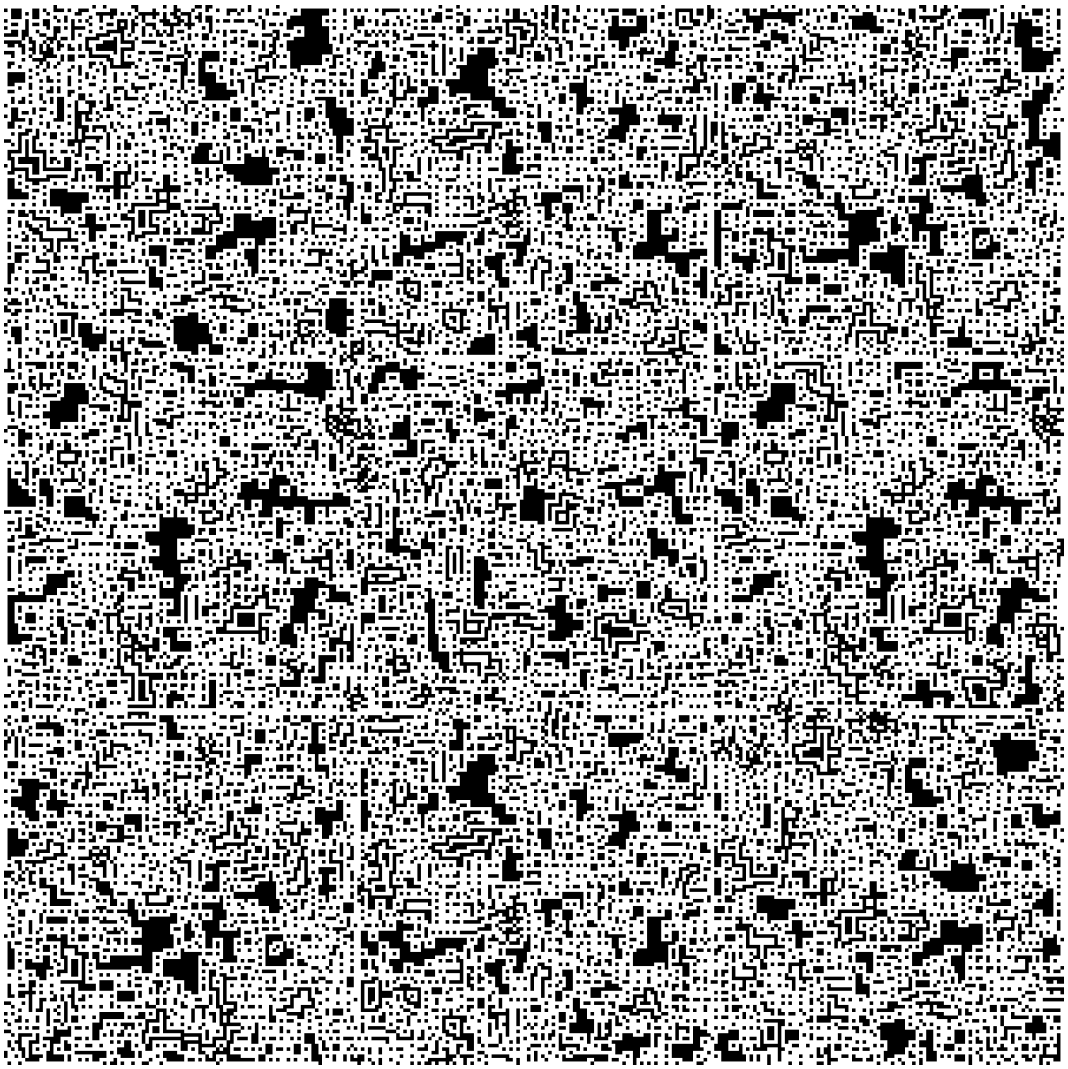
Rysunek 4.17: $C = [[' >=', 0.131], [' >=', 0.916], [' <=', 0.037]]$



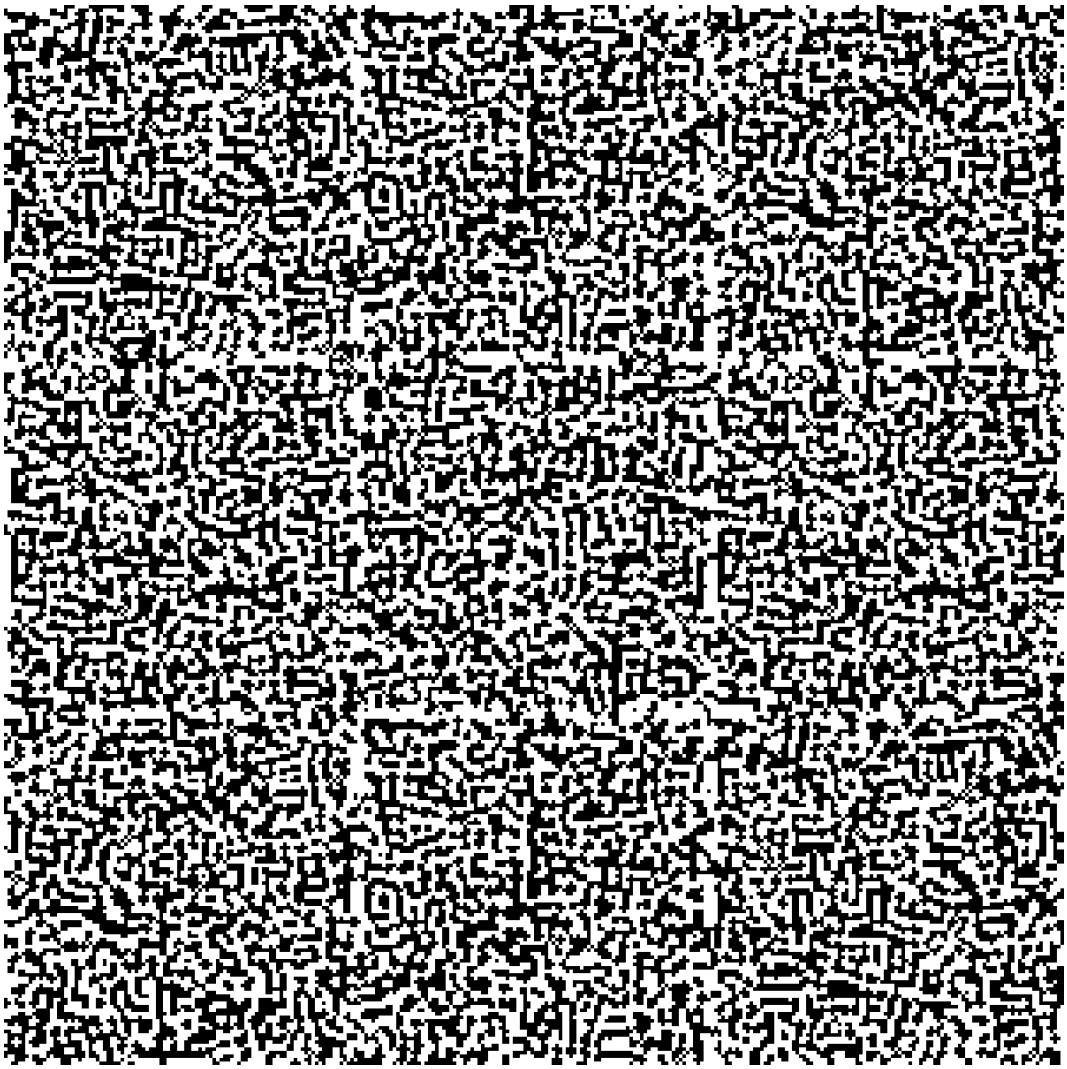
Rysunek 4.18: $C = [['<=', 0.860], ['>', 0.956], ['>', 0.917], ['<', 0.337]]$



Rysunek 4.19: $C = [['>', 0.901], ['>', 0.694], ['<=', 0.184], ['<=', 0.217]]$



Rysunek 4.20: $C = [['<=', 0.007], ['>', 0.972], ['<=', 0.184], ['<', 0.347]]$



Rysunek 4.21: $C = [['<=', 0.007], ['<=', 0.358], ['>', 0.787], ['<=', 0.089]]$

Rozdział 5.

Zakończenie

W ramach prezentowanej pracy, w rozdziale 2. omówione zostało pojęcie proceduralnego generowania treści wraz z jego historią, przykładowymi aplikacjami w branży gier komputerowych, potencjalnymi zagrożeniami wynikającymi ze stosowania powyższej metody oraz jej udziałem w budowaniu labiryntów. Rozdział 3. traktował natomiast o zastosowanych w projekcie technikach w postaci automatów komórkowych i algorytmów ewolucyjnych. Zostały tam opisane ich ogólne koncepcje wraz z pojęciami przydatnymi do zrozumienia części praktycznej zawartej w rozdziale 4.. W tym fragmencie przedstawiony został opis implementacji programu, przeprowadzonych nim badań, wyników w nich uzyskanych i płynących z nich wniosków.

Wnioski te prowadzą do konkluzji, że sposób dynamicznego generowania labiryntów metodą automatu komórkowego jest jak najbardziej skuteczny, a metoda szukania optymalnych parametrów drogą ewolucji jest obiecująca. Jednak niewątpliwie wymaga ona dalszego dopracowania.
We wnioskach nie zostały zawarte statystki odnoszące się do czasu generowania poszczególnych kafelków dla różnych konfiguracji parametrów, ponieważ nie było to przedmiotem badań, a implementacja w języku Python nie należy do optymalnych w tym względzie.

Na potencjalne rozwinięcia projektu składają się m.in. wzbogacenie procesu ewolucji o metodę symulacyjnej formy ewaluacji genotypów oraz dodatkowe bezpośrednie funkcje oceny w celu doprecyzowania oczekiwań wobec generowanych map. Uzyskane dotąd rezultaty mogą stanowić podstawę do dalszego kontynuowania prac nad budowaniem labiryntów metodą automatów komórkowych, dając badaczom lepszą świadomość jej możliwości. Zaprezentowana praca może być nie tylko bazą do dalszych badań nad generowaniem tytułowych jaskiń. Może posłużyć również jako fundament do budowania ich trójwymiarowej wersji zasugerowanej choćby przez Marka i wsp. w [14]. Projekt ten, bez względu na charakter usprawnień, może znaleźć zastosowanie jako podstawa do gry komputerowej lub jako narzędzie do testowania algorytmów stada.

Bibliografia

[1] Chad Adams. *Evolving Cellular Automata Rules for Maze Generation*. PhD thesis, 2018.

[2] Néstor Romeral Andrés. *Yavalath & Co. v1.0*. 2014.

[3] Hans-Georg Beyer. Evolution strategies. *Scholarpedia*, 2(8):1965, 2007.

[4] Michael Booth. The ai systems of left 4 dead. In *Artificial Intelligence and Interactive Digital Entertainment Conference at Stanford, 2009*, 2009.

[5] RogueBasin Community. Cellular automata method for generating random cave-like levels. `http://www.roguebasin.com/index.php?title=Cellular_Automata_Method_for_Generating_Random_Cave-Like_Levels#The_isolated_cave_problem`, 2016.

[6] Tom Forsyth. Game programming gems 3, chapter 2.6 cellular automata for physical modelling. Charles River Media, Inc., 2002.

[7] Frontier. Games by frontier-elite. `https://web.archive.org/web/20100127094607/http://frontier.co.uk/games/elite`, 27.01.2010.

[8] Jan Gorodkin, A Sørensen, and Ole Winther. Neural networks and cellular automata complexity. *Complex Systems*, 7(1):1–24, 1993.

[9] Erin J Hastings, Ratan K Guha, and Kenneth O Stanley. Demonstrating automatic content generation in the galactic arms race video game. In *AIIDE*, 2009.

[10] Erin J Hastings, Ratan K Guha, and Kenneth O Stanley. Evolving content in the galactic arms race video game. In *2009 IEEE Symposium on Computational Intelligence and Games*, pages 241–248. IEEE, 2009.

[11] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 9(1):1, 2013.

[12] Lawrence Johnson, Georgios N Yannakakis, and Julian Togelius. Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, page 10. ACM, 2010.

[13] Tobias Mahlmann, Julian Togelius, and Georgios N Yannakakis. Spicing up map generation. In *European Conference on the Applications of Evolutionary Computation*, pages 224–233. Springer, 2012.

[14] Benjamin Mark, Tudor Berechet, Tobias Mahlmann, and Julian Togelius. Procedural generation of 3d caves for games on the gpu. In *FDG*, 2015.

[15] Melanie Mitchell, James P. Crutchfield, and Rajarshi Das. Evolving cellular automata with genetic algorithms: A review of recent work. *First Int. Conf. on Evolutionary Computation and Its Applications*, 1, 05 2000.

[16] Jacob Olsen. Realtime procedural terrain generation. 2004.

[17] Barney Pell. Metagame in symmetric chess-like games. 1992.

[18] Elizabeth Reid. Cultural formations in text-based virtual realities. 1994.

[19] Noor Shaker, Miguel Nicolau, Georgios N Yannakakis, Julian Togelius, and Michael O'neill. Evolving levels for super mario bros using grammatical evolution. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 304–311. IEEE, 2012.

[20] Noor Shaker, Julian Togelius, and Mark J Nelson. *Procedural content generation in games*. Springer, 2016.

[21] Penelope Sweetser and Janet Wiles. Combining influence maps and cellular automata for reactive game agents. In *International Conference on Intelligent Data Engineering and Automated Learning*, pages 524–531. Springer, 2005.

[22] Julian Togelius, Renzo De Nardi, and Simon M Lucas. Towards automatic personalised content creation for racing games. In *2007 IEEE Symposium on Computational Intelligence and Games*, pages 252–259. IEEE, 2007.

[23] Julian Togelius, Emil Kastbjerg, David Schedl, and Georgios N Yannakakis. What is procedural content generation?: Mario on the borderline. In *Proceedings of the 2nd international workshop on procedural content generation in games*, page 3. ACM, 2011.

[24] Julian Togelius, Mike Preuss, Nicola Beume, Simon Wessing, Johan Hagelbäck, and Georgios N Yannakakis. Multiobjective exploration of the starcraft map space. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 265–272. IEEE, 2010.

[25] Glenn R Wichman. A brief history of rogue. `https://web.archive.org/web/20150217024917/http://www.wichman.org/roguehistory.html`, 1997.

[26] Wikipedia. Procedural generation. `https://en.wikipedia.org/wiki/Procedural_generation`, 21.08.2019.