Sztuczna inteligencja : Uczenie maszynowe : Genetyczne i awaryjne

(XII / XVI)

ĆWICZENIA

1. Algorytm genetyczny ma wspierać poszukiwanie różnorodności genetycznej wraz z przetrwaniem ważnych umiejętności (reprezentowanych przez wzorce genetyczne) dla domeny problemowej. Opisz, w jaki sposób różne operatory genetyczne mogą jednocześnie wspierać oba te cele.

2. Omów problem projektowania reprezentacji dla operatorów genetycznych w celu poszukiwania rozwiązań w różnych dziedzinach? Jaka jest tutaj rola błędu indukcyjnego?

3. Rozważ problem satysfakcji CNF w sekcji 12.1.3. W jaki sposób rola liczby dysjunkcji w wyrażeniu CNF wpływa na przestrzeń rozwiązań? Rozważ inne możliwe reprezentacje i operatory genetyczne dla problemu satysfakcji CNF. Czy możesz zaprojektować inny środek fitness?

4. Zbuduj algorytm genetyczny w języku do rozwiązania problemu satysfakcji CNF.

5. Rozważmy problem podróżującego sprzedawcy w Sekcji 12.1.3. Omów problem doboru odpowiedniej reprezentacji dla tego problemu. Zaprojektuj inne odpowiednie operatory genetyczne i środki przystosowania do tego problemu.

6. Zbuduj algorytm genetyczny do poszukiwania rozwiązania problemu podróżującego sprzedawcy.

7. Omów rolę technik, takich jak szare kodowanie, w kształtowaniu przestrzeni poszukiwań algorytmu genetycznego. Opisz dwa inne obszary, w których podobne techniki mogą być ważne.

8. Przeczytaj twierdzenie o schemacie Hollanda. W jaki sposób teoria schematów Hollanda opisuje ewolucję przestrzeni rozwiązań GA? Co to ma do powiedzenia na temat problemów, które nie są zakodowane jako ciągi bitów?

9. Jak ma się algorytm Bucket Brigade do algorytmu wstecznej propagacji?

10. Napisz program do rozwiązania problemu trzeciego prawa ruchu Keplera, opisanego ze wstępnym opisem przedstawionym w sekcji 12.2.2.

11. Omów ograniczenia (przedstawione w rozdziale 12.2.2) w stosowaniu technik programowania genetycznego do rozwiązywania problemów. Na przykład, które elementy rozwiązania nie mogą ewoluować w ramach paradygmatu programowania genetycznego?

12. Przeczytaj wczesną dyskusję na temat Gry w życie w kolumnie Gardnera w Scientific American . Omów inne konstrukcje ratunkowe, podobne do szybowca, w sekcji 12.3.1.

13. Napisz program a-life, który implementuje funkcjonalność przedstawioną na rysunkach 10-13.

14. Obszar badań agentowych został wprowadzony w sekcji 12.3. Zalecamy dalsze zapoznanie się z każdym z wymienionych projektów, a zwłaszcza badaniami Brooksa, Nilssona i Bensona lub Crutchfielda i Mitchella. Napisz krótki artykuł na jeden z tych tematów.

15. Omów rolę błędu indukcyjnego w reprezentacjach, strategiach wyszukiwania i operatorach stosowanych w modelach uczenia się przedstawionych w tej Części 12. Czy można rozwiązać ten problem? To znaczy, czy genetyczny model uczenia się działa wyłącznie ze względu na swoje założenia reprezentacyjne, czy też można go przełożyć na szersze dziedziny?

UCZENIE MASZYNOWE: GENETYCZNE I AWARYJNE

12.0 Społeczne i powstające modele uczenia się

Tak jak sieci koneksjonistyczne otrzymały wiele wczesnego wsparcia i inspiracji z celu stworzenia sztucznego systemu neuronowego, tak i szereg innych biologicznych analogii wpłynęło na projektowanie algorytmów uczenia maszynowego. W tym rozdziale omówiono algorytmy uczenia się wzorowane na procesach leżących u podstaw ewolucji: kształtowaniu populacji osobników poprzez przetrwanie jej najlepiej przystosowanych członków. Siła selekcji w populacji różnych osobników została wykazana w pojawieniu się gatunków w naturalnej ewolucji, a także w procesach społecznych leżących u podstaw zmiany kulturowej. Zostało również sformalizowane poprzez badania nad automatami komórkowymi, algorytmami genetycznymi, programowaniem genetycznym, sztucznym życiem i innymi formami nowych obliczeń.

Pojawiające się modele uczenia się symulują najbardziej elegancką i potężną formę adaptacji natury: ewolucję form życia roślin i zwierząt. Karol Darwin widział „… nie ma ograniczeń dla tej mocy powolnego i pięknego dostosowywania każdej formy do najbardziej złożonych relacji życiowych…". Dzięki temu prostemu procesowi wprowadzania odmian do kolejnych pokoleń i selektywnej eliminacji osób mniej sprawnych, w populacji pojawiają się adaptacje zwiększające możliwości i różnorodność. Ewolucja i wyłanianie się mają miejsce w populacjach osób wcielonych, których działania wpływają na innych, a na które z kolei wpływają inni. Zatem presje wybiórcze pochodzą nie tylko ze środowiska zewnętrznego, ale także z interakcji między członkami populacji. Ekosystem składa się z wielu członków, z których każdy ma role i umiejętności odpowiednie do własnego przetrwania, ale co ważniejsze, których skumulowane zachowanie kształtuje i jest kształtowane przez resztę populacji. Ze względu na swoją prostotę procesy leżące u podstaw ewolucji okazały się niezwykle ogólne. Ewolucja biologiczna tworzy gatunki, wybierając spośród zmian w genomie. Podobnie ewolucja kulturowa tworzy wiedzę, działając na przekazywanych społecznie i zmodyfikowanych jednostkach informacji. Algorytmy genetyczne i inne formalne analogi ewolucyjne tworzą coraz bardziej wydajne rozwiązania problemów, działając na populacjach potencjalnych rozwiązań problemów. Kiedy algorytm genetyczny jest używany do rozwiązywania problemów, ma on trzy odrębne etapy: po pierwsze, indywidualne potencjalne rozwiązania domeny problemowej są kodowane w reprezentacje, które obsługują niezbędne operacje zmiany i selekcji; często te reprezentacje są tak proste, jak ciągi bitów. W drugim etapie algorytmy kojarzenia i mutacji, analogiczne do aktywności seksualnej biologicznych form życia, tworzą nowe pokolenie osobników, które rekombinują cechy swoich rodziców. Wreszcie funkcja sprawności ocenia, które osoby są „najlepszymi" formami życia, to znaczy są najbardziej odpowiednie dla ostatecznego rozwiązania problemu. Osobom tym preferuje się przetrwanie i rozmnażanie, kształtując następną generację potencjalnych rozwiązań. Ostatecznie pokolenie osób zostanie zinterpretowane z powrotem do pierwotnej domeny problemu jako rozwiązania problemu. Algorytmy genetyczne są również stosowane do bardziej złożonych reprezentacji, w tym reguł produkcyjnych, w celu opracowania zestawów reguł przystosowanych do interakcji ze środowiskiem. Na przykład

programowanie genetyczne łączy i mutuje fragmenty kodu komputerowego, próbując rozwinąć program do rozwiązywania problemów, takich jak wychwytywanie niezmienników w zestawach danych. Przykład uczenia się jako interakcji społecznej prowadzącej do przetrwania można znaleźć w grach takich jak The Game of Life, pierwotnie stworzonych przez matematyka Johna Hortona Conwaya i przedstawionych szerszej społeczności przez Martina Gardnera w Scientific American. W tej grze narodziny, przetrwanie lub śmierć jednostek są funkcją ich własnego stanu i stanu ich bliskich sąsiadów. Zwykle do zdefiniowania gry wystarcza niewielka liczba reguł, zwykle trzy lub cztery. Pomimo tej prostoty, eksperymenty z grą w życie wykazały, że jest ona zdolna do ewolucji struktur o niezwykłej złożoności i zdolności, w tym samoreplikujących się wielokomórkowych „organizmów". Ważnym podejściem do sztucznego życia lub życia jest symulacja warunków ewolucji biologicznej poprzez interakcje maszyn skończonych, wraz z zestawami stanów i regułami przejścia. Te automaty są w stanie przyjmować informacje spoza nich samych, w szczególności od najbliższych sąsiadów. Ich zasady przejścia obejmują instrukcje dotyczące narodzin, kontynuowania życia i umierania. Kiedy populacja takich automatów zostaje uwolniona w domenie i pozwala działać jako równolegle asynchroniczni współpracujący agenci, czasami jesteśmy świadkami ewolucji pozornie niezależnych „form życia". Jako kolejny przykład, Rodney Brooks i jego uczniowie zaprojektowali i zbudowali proste roboty, które współdziałają jako autonomiczni agenci rozwiązujący problemy w warunkach laboratoryjnych. Nie ma centralnego algorytmu sterowania; raczej współpraca jawi się jako artefakt rozproszonych i autonomicznych interakcji jednostek. Społeczność a-life organizuje regularne konferencje i czasopisma odzwierciedlające ich pracę. W części 12.1 przedstawiamy modele ewolucyjne lub oparte na biologii z algorytmami genetycznymi , podejście do uczenia się, które wykorzystuje równoległość, wzajemne interakcje i często reprezentację na poziomie bitowym. W sekcji 12.2 przedstawiamy systemy klasyfikatorów i programowanie genetyczne, stosunkowo nowe obszary badawcze, w których techniki z algorytmów genetycznych są stosowane do bardziej złożonych reprezentacji, takich jak tworzenie i udoskonalanie zestawów reguł produkcji oraz tworzenie i dostosowywanie komputerów programy. W sekcji 12.3 przedstawiamy sztuczne życie. Rozpoczynamy 12.3 od wprowadzenia do „Gry w życie". Kończymy przykładem wyłaniającego się zachowania z badań w Santa Fe Institute.

12.1 Algorytm genetyczny

Podobnie jak sieci neuronowe, algorytmy genetyczne opierają się na metaforze biologicznej: postrzegają uczenie się jako konkurencję między populacją ewoluujących kandydatów do rozwiązań problemów. Funkcja „sprawności" ocenia każde rozwiązanie, aby zdecydować, czy przyczyni się ono do następnej generacji rozwiązań. Następnie, poprzez operacje analogiczne do transferu genów w rozmnażaniu płciowym, algorytm tworzy nową populację kandydujących rozwiązań. Niech P(t) zdefiniuje populację rozwiązań kandydujących x^t_i w czasie t:

$$P(t) = \{x^t_1, x^t_2, \ldots x^t_n\}$$

Przedstawiamy teraz ogólną postać algorytmu genetycznego:

procedure genetic algorithm;

begin

set time t:= 0;

initialize the population P(t);

while the termination condition is not met do

begin

evaluate fitness of each member of the population P(t);

select members from population P(t) based on fitness;

produce the offspring of these pairs using genetic operators;

replace, based on fitness, candidates of P(t), with these offspring;

set time t := t +1

end

end.

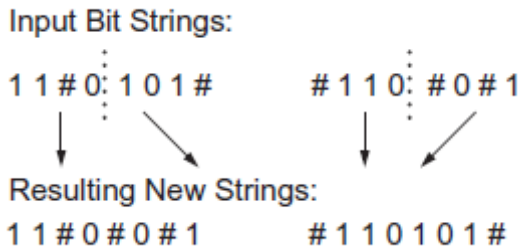
Ten algorytm określa podstawowe ramy uczenia się genetycznego; konkretne implementacje algorytmu tworzą instancję tego frameworka na różne sposoby. Jaki procent populacji zostaje zachowany? Jaki procent kojarzy się i rodzi potomstwo? Jak często i do kogo stosuje się operatory genetyczne? Procedura „zastąpienie najsłabszych kandydatów z P (t)" może zostać wdrożona w prosty sposób, eliminując określony procent najsłabszych kandydatów. Bardziej wyrafinowane podejścia mogą uporządkować populację według przystosowania, a następnie powiązać miarę prawdopodobieństwa eliminacji z każdym składnikiem, gdzie prawdopodobieństwo eliminacji jest funkcją odwrotną do jego przystosowania. Następnie algorytm zamiany wykorzystuje tę miarę jako czynnik przy wyborze kandydatów do wyeliminowania. Chociaż prawdopodobieństwo eliminacji byłoby bardzo niskie dla najsilniejszych członków społeczeństwa, istnieje szansa, że nawet najlepsze osobniki mogłyby zostać usunięte. Zaletą tego schematu jest to, że może on uratować niektóre osoby, których ogólna sprawność jest słaba, ale które zawierają jakiś element, który może przyczynić się do lepszego rozwiązania. Ten zastępczy algorytm ma wiele nazw, w tym Monte Carlo, proporcjonalny wybór sprawności i koło ruletki. Chociaż przykłady z sekcji 12.1.3 wprowadzają bardziej złożone reprezentacje, wprowadzimy kwestie reprezentacji związane z algorytmami genetycznymi przy użyciu prostego bitu ciągi reprezentujące rozwiązania problemów. Na przykład załóżmy, że chcemy, aby algorytm genetyczny nauczył się klasyfikować ciągi jedynek i zer. Możemy przedstawić populację ciągów bitowych jako wzorzec 1, 0 i #, gdzie # to „nie obchodzi", które może pasować do 0 lub 1. Zatem wzór 1 ## 00 ## 1 reprezentuje wszystkie ciągi ośmiu bitów, które zaczynają się i kończą 1 i które mają dwa 0 w środku. Algorytm genetyczny inicjalizuje P(0) do populacji wzorców kandydatów. Zazwyczaj początkowe populacje są wybierane losowo. Ocena rozwiązań kandydujących zakłada funkcję dopasowania $f(x_i^t)$, która zwraca miarę dopasowania kandydata w czasie t. Wspólna miara sprawności kandydata sprawdza go na zbiorze instancji szkoleniowych i zwraca procent poprawnych klasyfikacji. Korzystając z takiej funkcji przydatności, ocena nadaje każdemu kandydatowi rozwiązanie wartość:

$$f(x_i^t)/m(P, t)$$

gdzie m (P, t) jest średnią sprawnością wszystkich członków populacji. Często miara przydatności zmienia się w czasie, w związku z czym przydatność może być funkcją etapu ogólnego rozwiązania problemu, lub $f(x_i^t)$

Po ocenie każdego kandydata algorytm wybiera pary do rekombinacji. Rekombinacja wykorzystuje operatory genetyczne do tworzenia nowych rozwiązań, które łączą elementy ich rodziców. Podobnie jak w przypadku naturalnej ewolucji, sprawność kandydata określa zakres do którego się rozmnaża, przy czym kandydaci o najwyższych ocenach mają większe prawdopodobieństwo reprodukcji. Jak już wspomniano, selekcja jest często probabilistyczna, gdzie słabszym członkom daje się mniejsze

prawdopodobieństwo reprodukcji, ale nie są one całkowicie eliminowane. To, że niektórzy mniej sprawni kandydaci przeżyją, jest ważne, ponieważ nadal mogą zawierać jakiś istotny składnik rozwiązania, na przykład część wzoru bitowego, a reprodukcja może wydobyć ten składnik. Istnieje wiele operatorów genetycznych, które produkują potomstwo z cechami rodziców; najczęstszym z nich jest crossover. Crossover bierze dwa kandydujące rozwiązania i dzieli je, zamieniając komponenty, aby stworzyć dwóch nowych kandydatów. Rysunek 1 ilustruje skrzyżowanie na ciągach bitowych o długości 8.



Operator dzieli je w środku i tworzy dwoje dzieci, których początkowy segment pochodzi od jednego rodzica, a którego ogon pochodzi od drugiego. Zauważ, że podzielenie rozwiązania kandydata w środku to arbitralny wybór. Ten podział może występować w dowolnym punkcie reprezentacji i rzeczywiście, ten punkt podziału może być losowo dostosowywany lub zmieniany podczas procesu rozwiązywania. Na przykład załóżmy, że klasa docelowa jest zbiorem wszystkich ciągów zaczynających się i kończących na 1. Oba ciągi nadrzędne na rysunku 1 poradziłyby sobie stosunkowo dobrze w tym zadaniu. Jednak pierwsze potomstwo byłoby znacznie lepsze niż którekolwiek z rodziców: nie miałoby żadnych fałszywych alarmów i nie rozpoznałoby mniejszej liczby ciągów, które faktycznie znajdowały się w klasie rozwiązania. Zauważ również, że jego rodzeństwo jest gorsze niż którekolwiek z rodziców i prawdopodobnie tak będzie wyeliminowane w ciągu następnych kilku pokoleń. Mutacja to kolejny ważny operator genetyczny. Mutacja bierze jednego kandydata i losowo zmienia pewien jej aspekt. Na przykład mutacja może losowo wybrać bit we wzorze i zmienić go, przełączając 1 na 0 lub #. Mutacja jest ważna, ponieważ populacja początkowa może wykluczyć istotny składnik rozwiązania. W naszym przykładzie, jeśli żaden członek populacji początkowej nie ma 1 na pierwszej pozycji, to krzyżowanie, ponieważ zachowuje pierwsze cztery bity rodzica jako pierwsze cztery bity dziecka, nie może stworzyć potomstwa, które to robi. Mutacja byłaby potrzebna do zmiany wartości tych bitów. Inne operatory genetyczne, na przykład inwersja, również mogą wykonać to zadanie, i opisano w sekcji 12.1.3. Algorytm genetyczny działa aż do spełnienia pewnych wymagań dotyczących zakończenia, takich jak posiadanie jednego lub więcej kandydatów, których przydatność przekracza pewien próg. W następnej sekcji podajemy przykłady kodowania algorytmów genetycznych, operatorów i ocen sprawności dla dwóch sytuacji: spełnienia ograniczenia CNF i problemów podróżującego sprzedawcy.

12.1.3 Dwa przykłady: satysfakcja CNF i podróżujący sprzedawca

Następnie wybieramy dwa problemy i omawiamy kwestie reprezentacji oraz funkcje sprawności odpowiednie do ich rozwiązania. Należy zwrócić uwagę na trzy rzeczy: po pierwsze, wszystkie problemy nie są łatwo lub naturalnie zakodowane jako reprezentacje na poziomie bitowym. Po drugie, operatorzy genetyczni muszą zachować kluczowe relacje w populacji, na przykład obecność i wyjątkowość wszystkich miast w ramach wycieczki dla podróżującego sprzedawcy. Na koniec omówimy ważny związek między funkcją (funkcjami) przystosowania dla stanów problemu a kodowaniem tego problemu. Problem spełnialności formy normalnej koniunkcji (CNF) jest prosty: wyrażenie zdań występuje w postaci normalnej koniunkcji, gdy jest to ciąg zdań połączonych relacją i ($\wedge$). Każda z tych klauzul ma postać dysjunkcji, lub ($\vee$), literałów. Na przykład, jeśli literały to a, b, c, d,

e i f, to wyrażenie (¬a ∨ c) ∧ (¬a ∨ c ∨ ¬e) ∧ (¬b ∨ c ∨ d ∨ ¬ e) ∧ (a ∨ ¬b ∨ c) ∧ (¬e ∨ f) jest w CNF. To wyrażenie jest koniunkcją pięciu klauzul, każda klauzula jest rozłączeniem dwóch lub więcej literałów. Propozycje i ich satysfakcję przedstawiliśmy w rozdziale 2. Omówiliśmy formę CNF wyrażeń zdaniowych i zaproponowaliśmy metodę redukcji wyrażeń do CNF, kiedy przedstawiliśmy wnioskowanie o rozdzielczości. Spełnialność CNF oznacza, że musimy znaleźć przypisanie prawdy lub fałszu (1 lub 0) do każdego z sześciu literałów, tak aby wyrażenie CNF miało wartość true. Czytelnik powinien potwierdzić, że jednym rozwiązaniem dla wyrażenia CNF jest przypisanie fałszu każdemu z a, b i e. Inne rozwiązanie ma e false i c true.

Naturalną reprezentacją problemu satysfakcji CNF jest ciąg sześciu bitów, każdy bit w kolejności, reprezentujący prawdę (1) lub fałsz (0) dla każdego z sześciu literałów, ponownie w kolejności a, b, c, d , e i f. Zatem: 1 0 1 0 1 0 wskazuje, że a, c i e są prawdą, a b, d i f są fałszem, a zatem przykładowe wyrażenie CNF jest fałszywe. Czytelnik może zbadać wyniki innych przypisań prawdy do literałów wyrażenia. Wymagamy, aby działania każdego operatora genetycznego dały potomstwo, które jest przypisaniem prawdy dla wyrażenia CNF, a zatem każdy operator musi wytworzyć sześciobitowy wzór przypisań prawdy. Ważnym rezultatem naszego wyboru reprezentacji wzoru bitowego dla wartości prawdy literałów wyrażenia CNF jest to, że każdy z omówionych do tej pory operatorów genetycznych pozostawi wynikowy wzorzec bitowy jako uzasadnione możliwe rozwiązanie. Oznacza to, że crossover i mutacja pozostawiają wynikowy ciąg bitów jako możliwe rozwiązanie problemu. Nawet inne, rzadziej używane operatory genetyczne, takie jak inwersja (odwrócenie kolejności bitów we wzorze sześciobitowym) lub wymiana (zamiana dwóch różnych bitów we wzorze), pozostawiają wynikowy wzór bitowy jako uzasadnione możliwe rozwiązanie problemu CNF. W rzeczywistości, z tego punktu widzenia, trudno wyobrazić sobie lepiej dopasowaną reprezentację niż wzorzec bitowy dla problemu satysfakcji CNF. Wybór funkcji dopasowania dla tej populacji ciągów bitów nie jest tak prosty. Z jednego punktu widzenia albo przypisanie wartości prawdy do literałów sprawi, że wyrażenie będzie prawdziwe lub też będzie fałszywe. Jeśli określone przypisanie powoduje, że wyrażenie jest prawdziwe, wówczas znajduje się rozwiązanie; w przeciwnym razie tak nie jest. Na pierwszy rzut oka trudno jest określić funkcję dopasowania, która może oceniać „jakość" ciągów bitowych jako potencjalne rozwiązania. Istnieje jednak kilka alternatyw. Należy zauważyć, że pełne wyrażenie CNF składa się z koniunkcji pięciu klauzul. W ten sposób możemy stworzyć system oceny, który pozwoli nam uszeregować potencjalne rozwiązania wzorców bitowych w zakresie od 0 do 5, w zależności od liczby klauzul, które spełnia wzorzec. Tak więc wzór:

1 1 0 0 1 0 ma sprawność fizyczną 1,

0 1 0 0 1 0 ma sprawność fizyczną 2,

0 1 0 0 1 1 ma sprawność fizyczną 3, i

1 0 1 0 1 1 ma kondycję 5 i jest rozwiązaniem.

Ten algorytm genetyczny oferuje rozsądne podejście do problemu satysfakcji CNF. Jedną z jego najważniejszych właściwości jest użycie ukrytego paralelizmu zapewnianego przez populację rozwiązań. Operatory genetyczne mają naturalne dopasowanie do tej reprezentacji. Wreszcie, poszukiwanie rozwiązań wydaje się w naturalny sposób pasować do równoległej strategii „dziel i rządź", ponieważ przydatność jest oceniana na podstawie liczby elementów problemu, które są spełnione. W ćwiczeniach zachęca się czytelnika do rozważenia innych aspektów tego problemu.

PRZYKŁAD 12.2.2: PROBLEM Z PODRÓŻĄCYM SPRZEDAWCĄ

Problem podróżującego sprzedawcy (TSP) jest klasyczny dla sztucznej inteligencji i informatyki. Wprowadziliśmy to wraz z omówieniem wykresów w Części 3. Jego pełna przestrzeń stanów wymaga uwzględnienia N! podaje, gdzie N to liczba miast do odwiedzenia. Okazało się, że jest NP-trudny, a wielu badaczy proponuje heurystyczne podejście do jego rozwiązania. Sformułowanie problemu jest proste: sprzedawca musi odwiedzić N miast w ramach trasy sprzedaży. Z każdą parą miast na trasie związany jest koszt (np. Przebieg, opłata za przelot). Znajdź najmniej kosztowną ścieżkę dla sprzedawcy, aby rozpocząć w jednym mieście, odwiedzić wszystkie inne miasta dokładnie raz i wrócić do domu. TSP ma kilka bardzo przydatnych zastosowań, w tym wiercenie płytek drukowanych, krystalografię rentgenowską i trasowanie w produkcji VLSI. Niektóre z tych problemów wymagają odwiedzenia dziesiątek tysięcy punktów (miast) przy minimalnej ścieżce kosztów. Bardzo interesującym pytaniem w analizie klasy problemów TSP jest to, czy warto przez wiele godzin uruchamiać kosztowną stację roboczą, aby uzyskać prawie optymalne rozwiązanie, czy też uruchomić tani komputer na kilka minut, aby uzyskać „wystarczająco dobre" wyniki dla tych aplikacji . TSP to interesujący i trudny problem z wieloma konsekwencjami dotyczącymi strategii wyszukiwania. Jak możemy użyć algorytmu genetycznego do rozwiązania tego problemu? Po pierwsze, wybór reprezentacji ścieżki odwiedzanych miast, a także stworzenie zestawu operatorów genetycznych dla tej ścieżki nie jest trywialne. Projekt funkcji fitness jest jednak bardzo proste: wszystko, co musimy zrobić, to oszacować koszt długości ścieżki. Moglibyśmy wtedy uporządkować ścieżki według ich kosztu, im tańsze tym lepsze. Rozważmy kilka oczywistych reprezentacji, które okazują się skomplikowane konsekwencje. Załóżmy, że mamy dziewięć miast do odwiedzenia, 1, 2, ..., 9, więc reprezentację ścieżki tworzymy jako uporządkowaną listę tych dziewięciu liczb całkowitych. Załóżmy, że po prostu uczynimy każde miasto czterobitowym wzorem 0001, 0010,. . . 1001. Zatem wzorzec: 0001 0010 0011 0100 0101 0110 0111 1000 1001 reprezentuje wizytę w każdym mieście w kolejności jego numeracji. Dodaliśmy spacje do ciągu tylko po to, aby był łatwiejszy do odczytania. A co z operatorami genetycznymi? Crossover jest zdecydowanie niedostępny, ponieważ nowy sznurek wyprodukowany przez dwoje różnych rodziców najprawdopodobniej nie reprezentowałby ścieżki, która odwiedza każde miasto dokładnie raz. W rzeczywistości, dzięki skrzyżowaniu, niektóre miasta mogą zostać usunięte, podczas gdy inne są odwiedzane więcej niż raz. A co z mutacją? Załóżmy, że skrajny lewy fragment szóstego miasta, 0110, jest zmieniony na 1? 1110 lub 14 nie jest już legalnym miastem. Odwrócenie i zamiana miast (cztery bity we wzorze miasta) w obrębie wyrażenia ścieżki byłyby dopuszczalnymi operatorami genetycznymi, ale czy byłyby one wystarczająco potężne, aby uzyskać zadowalające rozwiązanie? W rzeczywistości jednym ze sposobów spojrzenia na poszukiwanie minimalnej ścieżki byłoby wygenerowanie i ocena wszystkich możliwych permutacji N elementów listy miast. Operatorzy genetyczni muszą być w stanie wyprodukować wszystkie permutacje. Innym podejściem do TSP byłoby zignorowanie reprezentacji wzoru bitowego i nadanie każdemu miastu alfabetycznej lub numerycznej nazwy, np. 1, 2, ..., 9; ułóż ścieżkę przez miasta w kolejności tych dziewięciu cyfr, a następnie wybierz odpowiednie operatory genetyczne do tworzenia nowych ścieżek. Mutacja, o ile byłaby przypadkową wymianą dwóch miast na ścieżce, byłaby w porządku, ale operator skrzyżowania między dwiema ścieżkami byłby bezużyteczny. Wymiana fragmentów ścieżki z innymi fragmentami tej samej ścieżki lub dowolny operator, który tasował litery ścieżki (bez usuwania, dodawania lub powielania jakichkolwiek miast) działałby. Takie podejście utrudnia jednak łączenie w potomstwo „lepszych" elementów wzorców na drogach miast dwojga różnych rodziców. Wielu badaczy stworzyło operatory krzyżowania, które rozwiązują te problemy i pozwalają nam pracować z uporządkowaną listą odwiedzonych miast. Na przykład Davis zdefiniował operator o nazwie crossover zamówienia. Załóżmy, że mamy dziewięć miast, 1, 2, ..., 9, a kolejność liczb całkowitych reprezentuje kolejność odwiedzonych miast. Order crossover buduje potomstwo, wybierając podciąg miast na drodze jednego z rodziców. Zachowuje również względne uporządkowanie miast od drugiego rodzica. Najpierw wybierz dwa punkty cięcia oznaczone „|", które są losowo wstawiane w to samo miejsce

każdego z rodziców. Lokalizacje punktów cięcia są losowe, ale po wybraniu te same lokalizacje są używane dla obojga rodziców. Na przykład dla dwojga rodziców p1 i p2, z punktami odcięcia po trzecim i siódmym mieście:

p1 = (1 9 2 | 4 6 5 7 | 8 3)

p2 = (4 5 9 | 1 8 7 6 | 2 3)

dwoje dzieci c1 i c2 jest tworzone w następujący sposób. Najpierw segmenty między punktami cięcia są kopiowane do potomstwa:

c1 = (x x x | 4 6 5 7 | x x)

c2 = (x x x | 1 8 7 6 | x x)

Następnie, zaczynając od drugiego punktu odcięcia jednego z rodziców, miasta drugiego rodzica są kopiowane w tej samej kolejności, pomijając miasta już obecne. Po osiągnięciu końca sznurka kontynuuj od początku. Zatem sekwencja miast z p2 jest następująca:

2 3 4 5 9 1 8 7 6

Po usunięciu miast 4, 6, 5 i 7, ponieważ są one już częścią pierwszego dziecka, otrzymujemy skróconą listę 2, 3, 9, 1 i 8, która następnie uzupełnia, zachowując kolejność znalezioną w p2 pozostałe miasta do odwiedzenia przez c1:

c1 = (2 3 9 | 4 6 5 7 | 1 8)

W podobny sposób możemy stworzyć drugie dziecko c2:

c2 = (3 9 2 | 1 8 7 6 | 4 5)

Podsumowując, w kolejności krzyżowania się fragmenty ścieżki są przekazywane od jednego rodzica p1 do dziecka c1, podczas gdy kolejność pozostałych miast dziecka c1 jest dziedziczona po drugim rodzicu p2. Potwierdza to oczywistą intuicję, że kolejność miast będzie ważna w generowaniu najmniej kosztownej ścieżki, dlatego też bardzo ważne jest, aby informacje dotyczące zamawiania były przekazywane dzieciom przez sprawnych rodziców. Algorytm krzyżowania zamówień gwarantuje również, że dzieci będą miały legalne wycieczki, odwiedzając wszystkie miasta dokładnie raz. Gdybyśmy chcieli dodać operator mutacji do tego wyniku, musielibyśmy, jak wspomniano wcześniej, uważać, aby była to wymiana miast w obrębie ścieżka. Operator inwersji, po prostu odwracający kolejność wszystkich miast w wycieczce, nie zadziała (nie ma nowej ścieżki, gdy wszystkie miasta są odwrócone). Jeśli jednak kawałek w ścieżce zostanie wycięty i odwrócony, a następnie zastąpiony, byłoby to dopuszczalne użycie inwersji. Na przykład, używając cięcia | wskaźnik jak poprzednio, ścieżka:

c1 = (2 3 9 | 4 6 5 7 | 1 8),

ulega odwróceniu środkowej sekcji,

c1 = (2 3 9 | 7 5 6 4 | 1 8)

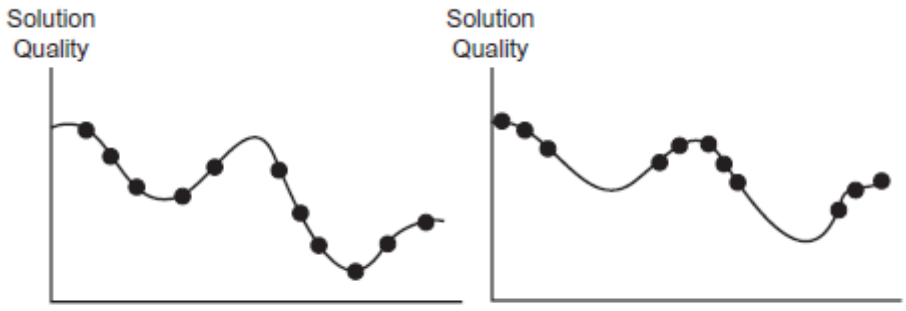
Można zdefiniować nowy operator mutacji, który losowo wybierze miasto i umieści je w nowej, losowo wybranej lokalizacji na ścieżce. Ten operator mutacji mógłby również działać na fragmencie ścieżki, na przykład, aby zająć podścieżkę trzech miast i umieścić je w tej samej kolejności w nowym miejscu na ścieżce. Inne sugestie są w ćwiczeniach.

12.1.4 Ocena algorytmu genetycznego

Powyższe przykłady podkreślają unikalne problemy algorytmu genetycznego dotyczące reprezentacji wiedzy, wyboru operatora i projektowania funkcji przystosowania. Wybrana reprezentacja musi wspierać operatory genetyczne. Czasami, tak jak w przypadku problemu satysfakcji CNF, reprezentacja poziomu bitów jest naturalna. W tej sytuacji tradycyjne genetyczne operatory krzyżowania i mutacji mogłyby zostać bezpośrednio wykorzystane do stworzenia potencjalnych rozwiązań. Zupełnie inną sprawą był problem podróżującego sprzedawcy. Po pierwsze, wydaje się, że nie ma żadnych naturalnych reprezentacji poziomu bitów dla tego problemu. Po drugie, należało wymyślić nowe operatory mutacji i krzyżowania, które zachowałyby własność potomstwa, która miała być legalną ścieżką przez wszystkie miasta, odwiedzając każde tylko raz. Wreszcie operatorzy genetyczni muszą przekazać „znaczące" informacje o potencjalnym rozwiązaniu następnemu pokoleniu. Jeśli ta informacja, podobnie jak w przypadku spełnialności CNF, jest przypisaniem wartości prawdy, operatorzy genetyczni muszą ją zachować w następnym pokoleniu. W przypadku TSP organizacja ścieżki była krytyczna, więc jak omówiliśmy, składniki tej ścieżki muszą być przekazywane potomkom. Ten udany transfer opiera się zarówno na wybranej reprezentacji, jak i na operatorach genetycznych zaprojektowanych dla każdego problemu. Przedstawienie pozostawiamy jeszcze jedną ostatnią kwestią, problemem „naturalności" wybranej reprezentacji. Załóżmy, że jako prosty, choć nieco sztuczny przykład, chcemy, aby nasze operatory genetyczne rozróżniały liczby 6, 7, 8 i 9. Reprezentacja liczb całkowitych daje bardzo naturalny i równomiernie rozłożony porządek, ponieważ w ramach dziesięciu liczb całkowitych o podstawie następny element jest po prostu o jeden więcej niż poprzedni. Jednak wraz ze zmianą na binarność ta naturalność znika. Rozważ wzory bitów dla 6, 7, 8 i 9: 0110 0111 1000 1001 Zwróć uwagę, że między 6 a 7 oraz między 8 a 9 występuje zmiana 1 bitu. Jednak między 7 a 8 wszystkie cztery bity się zmieniają! Ta reprezentacyjna anomalia może być ogromna w przypadku próby wygenerowania rozwiązania wymagającego jakiejkolwiek organizacji tych czterech wzorców bitowych. Szereg technik, zwykle w ramach ogólnego pojęcia szarego kodowania, rozwiązuje ten problem niejednolitej reprezentacji. Na przykład zakodowaną na szaro wersję pierwszych szesnastu liczb dwójkowych można znaleźć w tabeli 1. Zwróć uwagę, że każda liczba różni się dokładnie o jeden bit od swoich sąsiadów. Używając szarego kodowania zamiast standardowych liczb binarnych, przejścia operatora genetycznego między stanami bliskich sąsiadów są naturalne i płynne.

Binary	Gray
0000	0000
0001	0001
0010	0011
0011	0010
0100	0110
0101	0111
0110	0101
0111	0100
1000	1100
1001	1101
1010	1111
1011	1110
1100	1010
1101	1011
1110	1001
1111	1000

Ważną zaletą algorytmu genetycznego jest równoległy charakter jego poszukiwań. Algorytmy genetyczne wdrażają potężną formę wspinaczki górskiej, która utrzymuje wiele rozwiązań, eliminuje mało obiecujące i ulepsza dobre rozwiązania. Rysunek 2, dostosowany od Hollanda , pokazuje wiele rozwiązań zbiegających się w kierunku optymalnych punktów w przestrzeni poszukiwań.



a. The beginning search space

b. The search space after n generations

Na tym rysunku oś pozioma przedstawia możliwe punkty w przestrzeni rozwiązań, podczas gdy oś pionowa odzwierciedla jakość tych rozwiązań. Kropki na krzywej należą do aktualnej populacji potencjalnych rozwiązań algorytmu genetycznego. Początkowo rozwiązania są rozproszone w przestrzeni możliwych rozwiązań. Po kilku pokoleniach mają tendencję do skupiania się wokół obszarów o wyższej jakości rozwiązań. Kiedy opisujemy nasze poszukiwania genetyczne jako „wspinanie się po wzgórzach", pośrednio uznajemy poruszanie się po „krajobrazie fitness". Ten krajobraz będzie miał swoje doliny, szczyty, z lokalnymi maksimami i minimami. W rzeczywistości niektóre z nieciągłości w przestrzeni będą artefaktami reprezentacji i operatorów genetycznych wybranych dla problemu. Ta nieciągłość może być na przykład spowodowana brakiem szarego kodowania, jak właśnie omówiono. Zwróć też uwagę na to ,że algorytmy genetyczne, w przeciwieństwie do sekwencyjnych form wspinaczki górskiej, nie odrzucają od razu mało obiecujących rozwiązań. Dzięki operatorom genetycznym nawet słabe rozwiązania mogą nadal przyczyniać się do tworzenia przyszłych kandydatów. Inną różnicą między algorytmami genetycznymi a heurystykami przestrzeni stanów przedstawionymi w rozdziale 4 jest analiza różnicy stanu obecnego / stanu celu. Treść informacji wspierająca algorytm A*, wymagała oszacowania „wysiłku", aby przejść między stanem obecnym a stanem celu. Żadna taka miara nie jest wymagana w przypadku algorytmów genetycznych, po prostu pewna miara przydatności każdego z nich bieżące generowanie potencjalnych rozwiązań. Nie jest również wymagane ścisłe porządkowanie kolejnych stanów na liście otwartej, jak widzieliśmy podczas przeszukiwania przestrzeni stanów; istnieje raczej populacja pasujących rozwiązań do problemu, każdy potencjalnie dostępny, aby pomóc w tworzeniu nowych możliwych rozwiązań w ramach paradygmatu wyszukiwania równoległego. Ważnym źródłem mocy algorytmu genetycznego jest ukryta równoległość nieodłączna od operatorów ewolucyjnych. W porównaniu z przeszukiwaniem w przestrzeni stanów i uporządkowaną listą otwartą, wyszukiwanie przebiega równolegle, operując na całych rodzinach potencjalnych rozwiązań. Ograniczając reprodukcję słabszych kandydatów, algorytmy genetyczne mogą nie tylko wyeliminować to rozwiązanie, ale wszystkich jego potomków. Na przykład ciąg 101 # 0 ## 1, jeśli zostanie przerwany w środku, może być rodzicem całej rodziny ciągów w postaci 101 #____. Jeśli rodzic zostanie uznany za niezdolnego do życia, jego eliminacja może również usunąć całe potencjalne potomstwo i być może także możliwość rozwiązania problemu. Ponieważ algorytmy genetyczne są coraz szerzej wykorzystywane w rozwiązywaniu problemów stosowanych, a także w modelowaniu naukowym,

rośnie zainteresowanie próbami zrozumienia ich podstaw teoretycznych. Kilka pytań, które pojawiają się naturalnie, to:

1. Czy możemy scharakteryzować typy problemów, w przypadku których GA będą dobrze działać?

2. W przypadku jakich typów problemów radzą sobie słabo?

3. Co to w ogóle „oznacza" dla AH, aby działał dobrze lub źle dla typu problemu?

4. Czy są jakieś prawa, które mogą opisać makropoziom zachowania GA? W szczególności, czy istnieją jakieś przewidywania dotyczące zmian przystosowania podgrup populacji w czasie?

5. Czy istnieje sposób, aby opisać zróżnicowane skutki różnych operatorów genetycznych, krzyżowania się, mutacji, inwersji itp. W czasie?

6. W jakich okolicznościach (jakie problemy i jakie operatory genetyczne) GA będą działać lepiej niż tradycyjne metody wyszukiwania AI?

Zajęcie się wieloma z tych problemów znacznie wykracza poza zakres naszej książki. W rzeczywistości, jak wskazuje Mitchell, u podstaw algorytmów genetycznych wciąż istnieje więcej otwartych pytań niż ogólnie przyjętych odpowiedzi. Niemniej od początku pracy w GA próbowali to zrobić badacze, w tym Holland zrozumieć, jak działają GA. Chociaż zajmują się kwestiami na poziomie makro, takimi jak sześć właśnie zadanych pytań, ich analiza rozpoczyna się od przedstawienia na poziomie mikro lub bitowym. Holland (1975) wprowadził pojęcie schematu jako ogólnego wzorca i „budulca" rozwiązań. Schemat to wzorzec ciągów bitów opisany przez szablon składający się z 1, 0 i # (nie obchodzi mnie to). Na przykład schemat 1 0 # # 0 1 przedstawia rodzinę ciągów sześciobitowych rozpoczynających się od 1 0 i kończących się na 0 1. Ponieważ środkowy wzorzec # # opisuje cztery wzorce bitowe, 0 0, 0 1, 1 0, 1 1, cały schemat reprezentuje cztery wzorce po sześć jedynek i zer. Tradycyjnie mówi się, że każdy schemat opisuje hiperpłaszczyznę (Goldberg 1989); w tym przykładzie hiperpłaszczyzna przecina zestaw wszystkich możliwych reprezentacji sześciobitowych. Głównym założeniem tradycyjnej teorii AH jest to, że schematy są elementami składowymi rodzin rozwiązań. Mówi się, że genetyczni operatorzy krzyżowania i mutacji manipulują tymi schematami w kierunku potencjalnych rozwiązań. Specyfikacja opisująca tę manipulację nosi nazwę twierdzenia o schemacie (Holland 1975, Goldberg 1989). Według Holland, system adaptacyjny musi identyfikować, testować i uwzględniać właściwości strukturalne, zgodnie z hipotezą, aby zapewnić lepszą wydajność w pewnym środowisku. Schematy mają być formalizacją tych właściwości strukturalnych. Analiza schematu Hollanda sugeruje, że algorytm doboru przystosowania w coraz większym stopniu koncentruje się na podzbiorach przestrzeni poszukiwań o szacowanej najlepszej sprawności; to są podzbiory opisywane są schematami sprawności ponadprzeciętnej. Crossover operatora genetycznego łączy razem bloki budulcowe o wysokiej sprawności w tej samej strunie, próbując stworzyć coraz bardziej dopasowane struny. Mutacja pomaga zagwarantować, że różnorodność (genetyczna) nigdy nie zostanie usunięta z poszukiwań; to znaczy, że nadal badamy nowe elementy krajobrazu fitness. Algorytm genetyczny można zatem postrzegać jako napięcie między otwarciem ogólnego procesu wyszukiwania a uchwyceniem i zachowaniem ważnych (genetycznych) cech w tej przestrzeni poszukiwań. Chociaż oryginalna analiza Hollanda wyszukiwania GA skupiała się na poziomie bitowym, nowsze prace rozszerzyły tę analizę na alternatywne schematy reprezentacji (Goldberg 1989). W następnej sekcji zastosujemy techniki AH do bardziej złożonych reprezentacji,

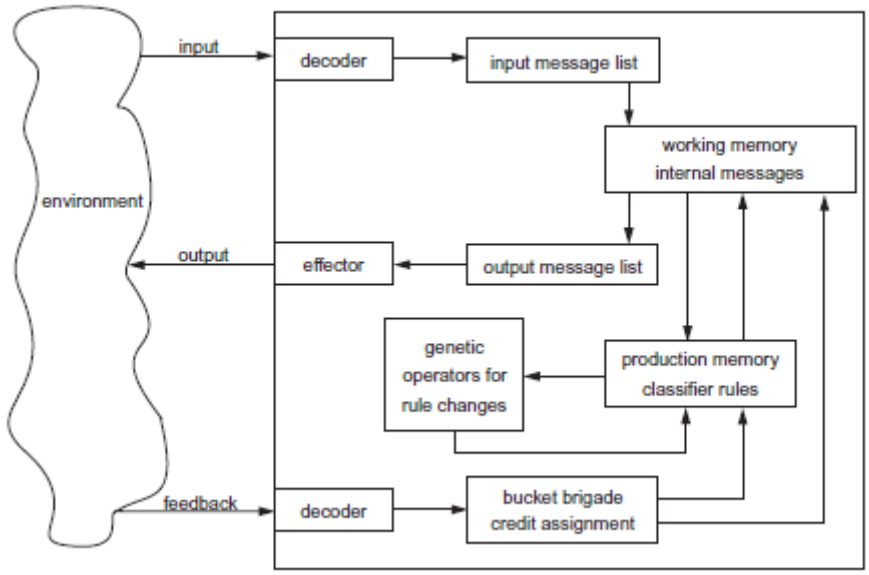
12.2 Systemy klasyfikacyjne i programowanie genetyczne

Wczesne badania algorytmów genetycznych skupiały się prawie wyłącznie na reprezentacjach niskiego poziomu, takich jak ciągi {0, 1, #}. Oprócz obsługi prostych instancji operatorów genetycznych, ciągi

bitów i podobne reprezentacje dają algorytmom genetycznym dużą moc innych podejść podsymbolicznych, takich jak sieci koneksjonistyczne. Istnieją jednak problemy, takie jak wędrowny sprzedawca, który ma bardziej naturalne kodowanie na bardziej złożonym poziomie reprezentacji. Możemy dalej zapytać, czy algorytmy genetyczne można zdefiniować dla jeszcze bogatszych reprezentacji, takich jak gdyby… wtedy… reguły lub fragmenty kodu komputerowego. Ważnym aspektem takich reprezentacji jest ich zdolność do łączenia odrębnych źródeł wiedzy wyższego poziomu poprzez łączenie reguł lub wywołania funkcji w celu spełnienia wymagania dotyczące konkretnego wystąpienia problemu. Niestety, trudno jest zdefiniować operatory genetyczne, które uchwycą składniową i semantyczną strukturę relacji logicznych, jednocześnie umożliwiając efektywne zastosowanie operatorów, takich jak crossover czy mutacja. Jednym z możliwych sposobów połączenia zdolności rozumowania reguł z uczeniem się genetyki jest przetłumaczenie zdań logicznych na ciągi bitowe i użycie standardowego operatora przecięcia. Niestety, w przypadku wielu tłumaczeń większość ciągów bitów wytwarzanych przez krzyżowanie i mutację nie będzie odpowiadać znaczącym zdaniom logicznym. Jako alternatywę dla przedstawiania rozwiązań problemów jako ciągów bitów, możemy zdefiniować odmiany crossovera, które można zastosować bezpośrednio do reprezentacji wyższego poziomu, na przykład jeśli ... to ... reguły lub fragmenty kodu w języku programowania wyższego poziomu. W tej sekcji omówiono przykłady każdego podejścia do rozszerzania mocy algorytmów genetycznych.

12.2.1 Systemy klasyfikatorów

Holland opracował architekturę rozwiązywania problemów zwaną systemami klasyfikatorów, która stosuje uczenie genetyczne do reguł w systemie produkcyjnym. System klasyfikatora zawiera znane elementy systemu produkcyjnego: reguły produkcji (zwane tutaj klasyfikatorami), pamięć roboczą, czujniki wejściowe (lub dekodery) i wyjścia (lub efektory).



Niezwykłe cechy systemu klasyfikatora obejmują wykorzystanie konkurencyjnego przetargu do rozwiązywania konfliktów, algorytmów genetycznych do uczenia się i algorytmu brygady wiaderkowej do przypisania zasługi i winy za reguły podczas nauki. Informacje zwrotne ze środowiska zewnętrznego umożliwiają ocenę sprawności kandydatów do klasyfikatorów, zgodnie z wymogami uczenia się genetyki. System klasyfikatora z rysunku 3 składa się z następujących głównych elementów:

1. Detektory komunikatów wejściowych z otoczenia.

2. Detektory komunikatów zwrotnych z otoczenia.

3. Efektory przenoszące wyniki aplikacji reguł z powrotem do środowiska.

4. Zestaw reguł produkcji składający się z populacji klasyfikatorów. Każdy klasyfikator ma przypisaną miarę sprawności.

5. Pamięć robocza dla reguł klasyfikatora. Ta pamięć integruje wyniki odpalania reguły produkcyjnej z informacjami wejściowymi.

6. Zbiór operatorów genetycznych do modyfikacji reguł produkcji.

7. System przypisywania uznania regułom tworzenia udanych działań

W rozwiązywaniu problemów klasyfikator działa jak tradycyjny system produkcyjny. Środowisko wysyła wiadomość, być może ruch w grze, do detektorów systemu klasyfikatora. To zdarzenie jest dekodowane i umieszczane jako wzorzec na wewnętrznej liście komunikatów, pamięci roboczej systemu produkcyjnego. Te komunikaty, w normalnym działaniu systemu produkcyjnego opartego na danych, są zgodne ze wzorcami warunków reguł klasyfikatora. O wyborze „najsilniej aktywowanych klasyfikatorów" decyduje schemat licytacji, w którym oferta jest funkcją zarówno skumulowanej sprawności klasyfikatora, jak i jakości dopasowania między bodźcem wejściowym a jego wzorcem stanu. Klasyfikatory z najbliższym dopasowaniem dodają komunikaty (działanie uruchomionych reguł) do pamięci roboczej. Zmieniona lista komunikatów może wysyłać komunikaty do efektorów, które działają na środowisko lub aktywują nowe reguły klasyfikatora w miarę kontynuacji przetwarzania systemu produkcyjnego. Systemy klasyfikatorów wdrażają formę uczenia się ze wzmocnieniem. Na podstawie informacji zwrotnej od nauczyciela lub funkcji oceny sprawności, uczący się oblicza przydatność populacji reguł kandydata i dostosowuje tę populację przy użyciu odmian uczenia się genetycznego. Systemy klasyfikatorów uczą się na dwa sposoby. Po pierwsze, istnieje system nagród, który dostosowuje mierniki sprawności reguł klasyfikatora, nagradzając pomyślne odpalenia reguł i karając za błędy. Algorytm przypisywania kredytów przekazuje część nagrody lub kary z powrotem do wszelkich reguł klasyfikatora, które przyczyniły się do uruchomienia ostatecznej reguły. To rozłożenie zróżnicowanych nagród między współpracującymi klasyfikatorami, a także tymi, które umożliwiły ich odpalenie, jest często realizowane w algorytmie brygady wiaderkowej. Algorytm brygady kubełkowej rozwiązuje problem przypisywania kredytu lub winy w sytuacjach, w których wyjście systemu może być iloczynem sekwencji odpalania reguły. W przypadku błędu, skąd mamy wiedzieć, którą regułę należy obwiniać? Czy odpowiedzialność za strzelanie ponosi ostatnia reguła, czy też jakaś poprzednia reguła, która dostarczyła błędnych informacji? Algorytm brygady wiaderkowej przypisuje zarówno zasługi, jak i winę do sekwencji zastosowań reguł zgodnie z miarami wkładu każdej reguły w ostateczny wniosek. Druga forma uczenia się modyfikuje same reguły za pomocą operatorów genetycznych, takich jak mutacja i krzyżowanie. Pozwala to na przetrwanie reguł odnoszących największe sukcesy i łączenie ich w celu tworzenia nowych klasyfikatorów, podczas gdy nieudane klasyfikatory reguł znikają. Każda reguła klasyfikatora składa się z trzech komponentów: warunek reguły pasuje do danych w pamięci roboczej w typowym sensie systemu produkcyjnego. Ucząc się, operatorzy genetyczni mogą modyfikować zarówno warunki, jak i działania reguł produkcji. Drugi składnik reguły, akcja, może skutkować zmianą wewnętrznej listy komunikatów (pamięci produkcyjnej). Wreszcie, każda reguła ma miarę sprawności. Ten parametr zmienia się, jak już wspomniano, zarówno przez udane, jak i nieudane działania. Środek ten jest pierwotnie przypisany do każdej reguły dotyczącej jej tworzenia przez operatorów genetycznych; na przykład można go ustawić jako średnią sprawność fizyczną jego dwojga rodziców. Prosty przykład ilustruje interakcje między tymi elementami systemu klasyfikatora. Załóżmy,

że zbiór obiektów do sklasyfikowania jest zdefiniowany przez sześć atrybutów (warunki c1, c2, ..., c6), a następnie przypuśćmy, że każdy z tych atrybutów może mieć pięć różnych wartości. Chociaż możliwe wartości każdego atrybutu są oczywiście różne (na przykład wartość c3 może być kolorem, podczas gdy c5 może opisywać pogodę), bez utraty ogólności nadamy każdemu atrybutowi wartość całkowitą z {1, 2, ..., 5}. Załóżmy, że warunki tych reguł umieszczają odpowiadający im obiekt w jednej z czterech klas: A1, A2, A3, A4. Bazując na tych ograniczeniach, każdy klasyfikator będzie miał postać: (c1 c2 c3 c4 c5 c6) → Ai, gdzie i = 1, 2, 3, 4. gdzie każdy ci we wzorze warunku oznacza wartość {1, 2,. .., 5} i-tego atrybutu warunku. Zwykle warunki mogą również przypisywać atrybutowi wartość # lub „nie obchodzi mnie to". Ai oznacza klasyfikację, A1, A2, A3 lub A4. Tabela 2 przedstawia zestaw klasyfikatorów. Należy zauważyć, że różne wzorce warunków mogą mieć tę samą klasyfikację, jak w regułach 1 i 2, lub te same wzorce, jak w regułach 3 i 5, mogą prowadzić do różnych klasyfikacji.

Condition (Attributes)	Action (Classification)	Rule Number
(1 # # # 1 #)	→ A1	1
(2 # # 3 # #)	→ A1	2
(# # 4 3 # #)	→ A2	3
(1 # # # # #)	→ A2	4
(# # 4 3 # #)	→ A3	5
etc.		

Jak opisano do tej pory, system klasyfikatorów jest po prostu kolejną formą wszechobecnego systemu produkcyjnego. Jedyną naprawdę nowatorską cechą reguł klasyfikatora w tym przykładzie jest użycie ciągów cyfr i # do reprezentowania wzorców warunków. To właśnie ta reprezentacja warunków ogranicza zastosowanie algorytmów genetycznych do reguł. Pozostała część dyskusji dotyczy uczenia się genów w systemach klasyfikatorów. Aby uprościć pozostałą część przykładu, weźmiemy pod uwagę tylko wydajność systemu klasyfikatora w uczeniu się klasyfikacji A1. Oznacza to, że zignorujemy inne klasyfikacje i przypiszemy wzorcom warunków wartość 1 lub 0 w zależności od tego, czy obsługują one klasyfikację A1, czy nie. Należy zauważyć, że w tym uproszczeniu nie ma utraty ogólności; można go rozszerzyć na problemy związane z uczeniem się więcej niż jednej klasyfikacji poprzez użycie wektora do wskazania klasyfikacji, które pasują do określonego wzorca warunkowego. Na przykład klasyfikatory z tabeli 12.2 można podsumować następująco:

(1 # # # 1 #) → (1 0 0 0)
(2 # # 3 # #) → (1 0 0 0)
(1 # # # # #) → (0 1 0 0)
(# # 4 3 # #) → (0 1 1 0)

W tym przykładzie ostatnie z tych podsumowań wskazuje, że atrybuty warunku obsługują reguły klasyfikacji A2 i A3, a nie A1 lub A4. Zastępując przypisanie 0 lub 1 tymi wektorami, algorytm uczący się może ocenić wydajność reguły w wielu klasyfikacjach. W tym przykładzie użyjemy reguł z tabeli 12.2 do wskazania prawidłowych klasyfikacji; zasadniczo będą funkcjonować jako nauczyciele lub osoby oceniające zgodność reguł w systemie uczenia się. Podobnie jak w przypadku większości uczących się genetyki, zaczynamy od przypadkowej populacji reguł. Każdy wzorzec stanu ma również przypisany parametr siły lub sprawności (liczba rzeczywista między 0,0 brak siły a 1,0 pełna siła. Ten parametr siły, s, jest obliczany na podstawie sprawności rodziców każdej reguły i mierzy jej przydatność historyczną . W każdym cyklu uczenia się reguły próbują klasyfikować dane wejściowe, a następnie są uszeregowane według nauczyciela lub miernika sprawności. Załóżmy na przykład, że w pewnym cyklu klasyfikator ma

następującą populację reguł klasyfikacji kandydatów, gdzie wynik 1 wskazuje że wzorzec prowadził do poprawnej klasyfikacji i 0, że nie:

(# # # 2 1 #) → 1 s = 0,6

(# # 3 # # 5) → 0 s = 0,5

(2 1 # # # #) → 1 s = 0,4

(# 4 # # # 2) → 0 s = 0,23

Załóżmy, że ze środowiska pochodzi nowa wiadomość wejściowa: (1 4 3 2 1 5), a nauczyciel (korzystając z pierwszej reguły z tabeli 12.2) klasyfikuje ten wektor wejściowy jako pozytywny przykład A1. Zastanówmy się, co się stanie, gdy pamięć robocza otrzyma ten wzorzec, a cztery kandydujące reguły klasyfikujące spróbują go dopasować. Dopasowanie zasad 1 i 2. Rozwiązywanie konfliktów odbywa się w drodze przetargu konkurencyjnego między dopasowanymi regułami. W naszym przykładzie cena ofertowa jest funkcją sumy dopasowań wartości atrybutów pomnożonych przez miarę siły reguły. Dopasowania typu „bez znaczenia" mają wartość 0,5, a dopasowania ścisłe mają wartość 1,0. Aby znormalizować, dzielimy ten wynik przez długość wektora wejściowego. Ponieważ wektor wejściowy pasuje do pierwszego klasyfikatora z dwoma dokładnymi i czterema „nie obchodzi", jego stawka wynosi ((4 * 0,5 + 2 * 1) * 0,6) / 6 lub 0,4. Drugi klasyfikator również pasuje do dwóch atrybutów i ma cztery „nie obchodzi", więc jego stawka wynosi 0,33. W naszym przykładzie uruchamia się tylko klasyfikator składający najwyższą ofertę, ale w bardziej skomplikowanych sytuacjach może być pożądane, aby procent ofert został zaakceptowany. Pierwsza reguła wygrywa i ogłasza swoje działanie, 1, wskazując, że ten wzorzec jest przykładem A1. Ponieważ ta czynność jest prawidłowa, miara przydatności przepisu 1 zostaje zwiększona do wartości od aktualnej do 1,0. Gdyby działanie tej zasady było nieprawidłowe, środek sprawności zostałby obniżony. Gdyby system wymagał wielokrotnego odpalenia zestawu reguł, aby wywołać jakiś skutek w środowisku, wszystkie reguły odpowiedzialne za ten wynik otrzymałyby pewną część nagrody. Dokładna procedura obliczania sprawności reguły różni się w zależności od systemu i może być dość złożona, obejmując użycie algorytmu brygady kubełkowej lub podobnej techniki przypisywania punktów. Szczegóły można znaleźć w Holland. Po obliczeniu przydatności reguł kandydata algorytm uczący się stosuje operatory genetyczne w celu utworzenia reguł następnej generacji. Najpierw algorytm wyboru określi najbardziej pasujących członków zestawu reguł. Ten wybór opiera się na mierniku sprawności, ale może również obejmować dodatkową wartość losową. Wartość losowa daje regułom o słabej przydatności możliwość reprodukcji, pomagając uniknąć zbyt pośpiesznej eliminacji reguł, które, choć ogólnie są słabe, mogą zawierać jakiś element pożądanego rozwiązania. Załóżmy, że pierwsze dwie reguły klasyfikacyjne w przykładzie są wybrane, aby przetrwać i rozmnażać się. Losowe wybieranie położenia zwrotnicy między czwartym i piątym elementem,

(# # # 2 | 1 #) → 1 s = 0,6

(# # 3 # | # 5) → 0 s = 0,5

produkuje potomstwo:

(# # 3 # | 1 #) → 0 s = 0,53

(# # # 2 | # 5) → 1 s = 0,57

Miara sprawności dzieci jest ważoną funkcją sprawności rodziców. Ważenie jest funkcją położenia punktu przecięcia. Pierwsze potomstwo ma 1/3 pierwotnego klasyfikatora 0,6 i 2/3 pierwotnego klasyfikatora 0,5. Zatem pierwsze potomstwo ma siłę (1/3 * 0,6) + (2/3 * 0,5) = 0,53. Przy podobnych

obliczeniach sprawność drugiego dziecka wynosi 0,57. Wynik odpalenia reguły klasyfikatora, zawsze 0 lub 1, idzie w parze z większością atrybutów, zachowując w ten sposób intuicję, że te wzorce są ważne w wynikach reguł. W typowym systemie klasyfikatorów te dwie nowe reguły, wraz z ich rodzicami, stanowiłyby podzbiór klasyfikatorów do działania systemu w następnym kroku czasowym. Można również zdefiniować operator mutacji. Prosta reguła mutacji polegałaby na losowej zmianie dowolnego wzorca atrybutu na inny prawidłowy wzorzec atrybutu; na przykład 5 może zostać zmutowane na 1, 2, 3, 4 lub #. Ponownie, jak zauważono w naszej dyskusji o GA, operatory mutacji są postrzegane jako wymuszające różnorodność w poszukiwaniu klasyfikatorów, podczas gdy crossover próbuje zachować i zbudować nowe dzieci z udanych fragmentów wzorców rodzicielskich. Nasz przykład był prosty i przeznaczony przede wszystkim do zilustrowania głównych elementów systemu klasyfikatora. W rzeczywistym systemie może zostać uruchomionych więcej niż jedna reguła, a każda z nich przekazuje wyniki do pamięci produkcyjnej. Często istnieje schemat podatkowy, który uniemożliwia klasyfikatorowi zajęcie zbyt dużego miejsca w procesie rozwiązywania problemu, obniżając jego przydatność za każdym razem, gdy wygrywa ofertę. Nie zilustrowaliśmy również algorytmu brygady wiaderkowej, który w różny sposób nagradza reguły wspierające pomyślne komunikaty wyjściowe do środowiska. Ponadto operatorzy genetyczni zwykle nie przerabiają klasyfikatorów przy każdej operacji systemu. Zamiast tego istnieje pewien ogólny parametr dla każdej aplikacji, który decyduje, być może, o analizie informacji zwrotnej ze środowiska, kiedy klasyfikatory powinny być ocenione i zastosowane operatory genetyczne. Wreszcie, nasz przykład pochodzi z systemów klasyfikacyjnych, które zaproponował Holland z University of Michigan. Podejście Michigan można postrzegać jako obliczeniowy model poznania, w którym wiedza (klasyfikatory) jednostki poznawczej jest wystawiona na działanie reagującego środowiska iw rezultacie podlega modyfikacjom w czasie. Oceniamy sukces całego systemu w czasie, przy czym znaczenie poszczególnych klasyfikatorów jest minimalne. Badano również alternatywne systemy klasyfikatorów, w tym prace na Uniwersytecie w Pittsburghu. Klasyfikator Pittsburgh koncentruje się na roli poszczególnych reguł w tworzeniu nowych generacji klasyfikatorów. Podejście to realizuje model uczenia indukcyjnego zaproponowany przez Michalskiego. W następnej sekcji rozważymy inną i szczególnie ekscytującą aplikację dla GA, ewolucję programów komputerowych.

12.2.2 Programowanie z operatorami genetycznymi

W kilku ostatnich podsekcjach widzieliśmy, jak GA są stosowane do coraz większych struktur reprezentacyjnych. To, co zaczęło się jako transformacja genetyczna na ciągach bitowych, przekształciło się w operacje na if. . . następnie . . . zasady. Można w naturalny sposób zapytać, czy można zastosować techniki genetyczne i ewolucyjne do produkcji innych narzędzi obliczeniowych na większą skalę. Były tego dwa główne przykłady: generowanie programów komputerowych i ewolucja systemów maszyn skończonych. Koza zasugerował, że udany program komputerowy może ewoluować poprzez kolejne zastosowania operatorów genetycznych. W programowaniu genetycznym adaptowane struktury są hierarchicznie zorganizowanymi segmentami programów komputerowych. Algorytm uczenia się utrzymuje populację programów kandydujących. Dopasowanie programu będzie mierzone jego zdolnością do rozwiązywania zestawu zadań, a programy są modyfikowane przez zastosowanie krzyżowania i mutacji w poddrzewach programu. Programowanie genetyczne przeszukuje przestrzeń programów komputerowych o różnej wielkości i złożoności; w rzeczywistości przestrzeń poszukiwań jest przestrzenią wszystkich możliwych programów komputerowych złożoną z funkcji i symboli terminali odpowiednich dla dziedziny problemu. Podobnie jak w przypadku wszystkich uczących się genetyki, to wyszukiwanie jest przypadkowe, w dużej mierze ślepe, a jednocześnie zaskakująco skuteczne. Programowanie genetyczne rozpoczyna się od początkowej populacji losowo generowanych programów składających się z odpowiednich fragmentów programu. Elementy te, odpowiednie dla dziedziny problemowej, mogą składać się ze standardowych operacji arytmetycznych,

innych powiązanych operacji programistycznych i funkcji matematycznych, a także funkcji logicznych i specyficznych dla dziedziny. Komponenty programu zawierają elementy danych typowych typów: logiczne, całkowite, zmiennoprzecinkowe, wektorowe, symboliczne lub wielowartościowe. Po inicjalizacji tysiące programów komputerowych jest hodowanych genetycznie. Produkcja nowych programów wiąże się z zastosowaniem operatorów genetycznych. Do produkcji programów komputerowych należy dostosować algorytmy krzyżowania, mutacji i innych algorytmów hodowlanych. Wkrótce zobaczymy kilka przykładów. Adekwatność każdego nowego programu jest następnie określana poprzez sprawdzenie, jak dobrze działa on w określonym środowisku problemowym. Charakter środka sprawności różni się w zależności od dziedziny problemu. Każdy program, który dobrze sobie radzi w tym zadaniu fitness, przetrwa, pomagając w wychowaniu dzieci następnego pokolenia. Podsumowując, programowanie genetyczne obejmuje sześć komponentów, z których wiele jest bardzo podobnych do wymagań dla GA:

1. Zbiór struktur podlegających transformacji przez operatory genetyczne.

2. Zestaw początkowych struktur dopasowanych do dziedziny problemowej.

3. Miara przydatności, ponownie zależna od dziedziny, do oceny struktur.

4. Zbiór operatorów genetycznych do transformacji struktur.

5. Parametry i opisy stanów, które opisują członków każdego pokolenia.

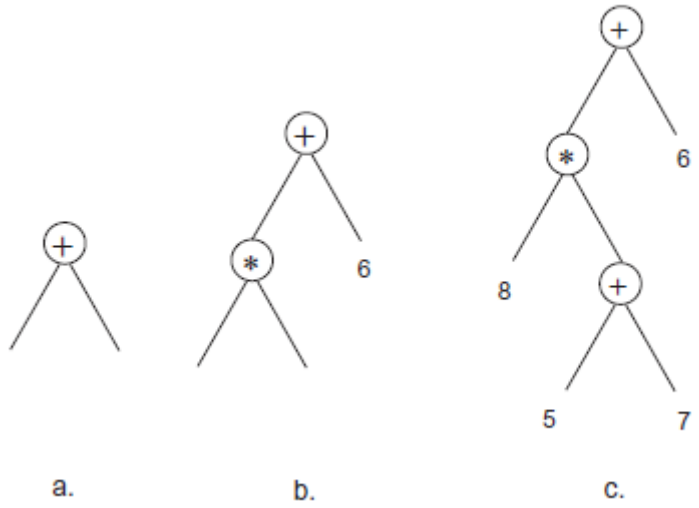
6. Zestaw warunków zakończenia.

W kolejnych akapitach szczegółowo omówimy każdy z tych tematów. Programowanie genetyczne manipuluje hierarchicznie zorganizowanymi modułami programu. Lisp był (i nadal pozostaje) podstawową reprezentacją komponentów języka programowania: Koza reprezentuje segmenty programu jako wyrażenia symboli Lispa lub wyrażenia s. Operatory genetyczne manipulują wyrażeniami s. W szczególności operatorzy odwzorowują struktury drzewiaste wyrażeń s (segmenty programu Lisp) na nowe drzewa (nowe segmenty programu Lisp). Chociaż to wyrażenie s jest podstawą wczesnych prac Kozy, inni badacze w większym stopniu zastosowali to podejście do różnych języków / paradygmatów programowania. Programowanie genetyczne stworzy użyteczne programy, biorąc pod uwagę, że dostępne są fragmenty atomowe i możliwe do oceny predykaty domeny problemowej. Kiedy ustawiamy domenę do generowania programów wystarczających do rozwiązania zestawu problemów, musimy najpierw przeanalizować, jakie terminale są wymagane dla jednostek w jej rozwiązaniu, a także jakie funkcje są niezbędne do wytworzenia tych terminali. Jak zauważa Koza „… użytkownik programowania genetycznego powinien wiedzieć… że pewien zestaw funkcji i terminali, które dostarcza, może przynieść rozwiązanie problemu”. Aby zainicjować struktury do adaptacji przez operatory genetyczne, musimy utworzyć dwa zbiory: F, zbiór funkcji i T, zbiór wartości końcowych wymaganych dla domeny. F może być tak proste, jak {+, *, -, /} lub może wymagać bardziej złożonych funkcji, takich jak sin (X), cos (X) lub funkcje dla operacji macierzowych. T może być liczbami całkowitymi, liczbami rzeczywistymi, macierzami lub bardziej złożonymi wyrażeniami. Symbole w T muszą być zamknięte pod funkcjami określonymi w F.

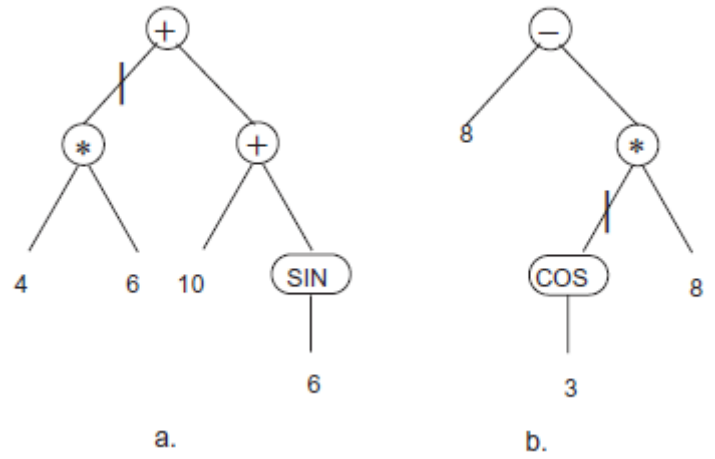
Następnie generowana jest populacja początkowych „programów” poprzez losowe wybieranie elementów ze związku zbiorów F i T. Na przykład, jeśli zaczniemy od wybrania elementu T, mamy zdegenerowane drzewo pojedynczego węzła głównego. Co ciekawsze, kiedy zaczniemy od elementu z F, powiedzmy +, otrzymamy węzeł główny drzewa z dwojgiem potencjalnych dzieci. Załóżmy, że inicjator następnie wybiera * (z dwojgiem potencjalnych dzieci) z F jako pierwsze dziecko, a następnie

terminal 6 z T jako drugie dziecko. Kolejny losowy wybór może dać terminal 8, a następnie funkcję + z F. Załóżmy, że kończy się wybierając 5 i 7 z T.

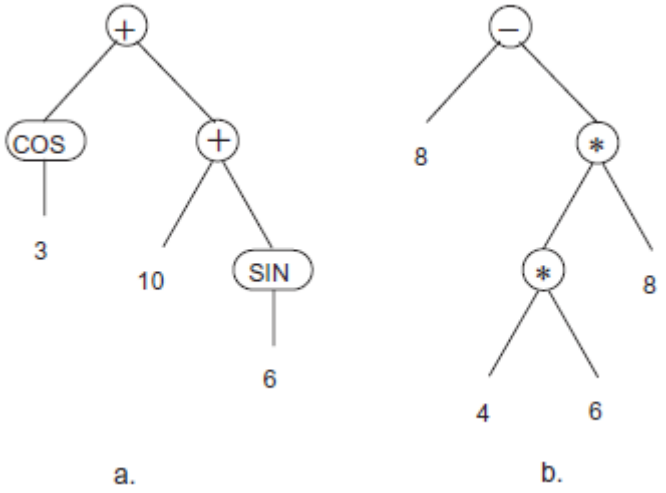
Program, który stworzyliśmy losowo, jest przedstawiony na rysunku 4.



Rysunek 4a przedstawia drzewo po pierwszym wybraniu +, 15.4b po wybraniu terminala 6 i 15.4c programu końcowego. Tworzona jest populacja podobnych programów w celu zainicjowania procesu programowania genetycznego. Zestawy ograniczeń, takie jak maksymalna głębokość rozwoju programów, mogą pomóc w przycinaniu tej populacji. Opis tych ograniczeń, a także różnych metod generowania populacji początkowych, można znaleźć u Kozy. Dyskusja do tej pory dotyczy kwestii reprezentacji (wyrażeń s) i zestawu struktur drzewiastych niezbędnych do zainicjowania sytuacji ewolucji programu. Następnie wymagamy miernika sprawności dla populacji programów. Miara sprawności zależy od dziedziny problemu i zwykle składa się z zestawu zadań, do których muszą się zmierzyć wyewoluowane programy. Sam miernik sprawności jest funkcją tego, jak dobrze każdy program radzi sobie z tymi zadaniami. Prosty surowy wynik sprawności dodałby różnice między tym, co wygenerował program, a wynikami wymaganymi przez rzeczywiste zadanie z domeny problemowej. W związku z tym surową przydatność można postrzegać jako sumę błędów w zestawie zadań. Oczywiście możliwe są inne środki sprawności. Dopasowanie znormalizowane dzieli surową przydatność przez całkowitą sumę możliwych błędów, a tym samym umieszcza wszystkie miary przystosowania w zakresie od 0 do 1. Normalizacja może mieć przewagę, gdy próbuje się dokonać wyboru z dużej populacji programów. Środek sprawności może również obejmować dostosowanie do rozmiaru programu, na przykład w celu nagradzania mniejszych, bardziej oszczędnych programów. Operatory genetyczne w programach obejmują zarówno transformacje samego drzewa, jak i wymianę struktur między drzewami. Koza (1992) opisuje pierwotne przemiany jako reprodukcję i krzyżowanie. Reprodukcja po prostu wybiera programy z obecnej generacji i kopiuje je (niezmienione) do następnej generacji. Crossover wymienia poddrzewa między drzewami reprezentującymi dwa programy. Na przykład, załóżmy, że pracujemy z dwoma programami nadrzędnymi z rysunku 5 i że losowe punkty są oznaczone | w rodzicu a i rodzicu b są wybierane do krzyżowania.



Wynikowe elementy potomne pokazano na rysunku 6.



Crossover może być również użyty do przekształcenia samotnego rodzica poprzez zamianę dwóch poddrzew od tego rodzica. Dwóch identycznych rodziców może stworzyć różne dzieci z losowo wybranymi punktami krzyżowania. Rdzeń programu można również wybrać jako punkt przecięcia. Istnieje wiele wtórnych i znacznie rzadziej używanych transformacji genetycznych drzew programu. Należą do nich mutacja, która po prostu wprowadza losowe zmiany w strukturach programu. Na przykład zastąpienie wartości końcowej inną wartością lub poddrzewem funkcji. Transformacja permutacyjna, podobnie jak operator inwersji na łańcuchach, działa również na pojedynczych programach, na przykład wymieniając symbole terminali lub poddrzewa. Stan rozwiązania odzwierciedla aktualna generacja programów. Nie ma zapisów dotyczących cofania się ani żadnej innej metody omijania krajobrazu fitness. Pod tym względem programowanie genetyczne jest bardzo podobne do algorytmu wspinania się po wzgórzu. Paradygmat programowania genetycznego odpowiada naturze, ponieważ ewolucja nowych programów jest procesem ciągłym. Niemniej jednak, z powodu braku nieskończonego czasu i obliczeń, ustalane są warunki zakończenia. Zwykle są one funkcją zarówno sprawności programu, jak i zasobów obliczeniowych. Fakt, że programowanie genetyczne jest techniką obliczeniowego generowania programów komputerowych, umieszcza ją w tradycji badań nad programowaniem automatycznym. Od najwcześniejszych dni sztucznej inteligencji naukowcy pracowali nad automatycznym tworzeniem programów komputerowych na podstawie fragmentarycznych informacji . Programowanie genetyczne można postrzegać jako kolejne narzędzie

w tej ważnej dziedzinie badawczej. Kończymy tę sekcję prostym przykładem programowania genetycznego zaczerpniętym z Mitchella.

PRZYKŁAD 3.2.1: EWOLUCJA PROGRAMU TRZECIEGO PRAWA RUCHU PLANETARNEGO KEPLERA

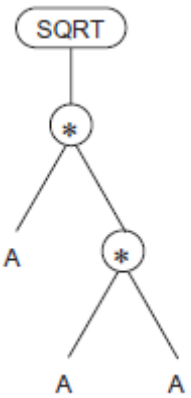
Koza opisuje wiele zastosowań programowania genetycznego do rozwiązywania interesujących problemów, ale większość z tych przykładów jest obszerna i zbyt złożona dla naszych obecnych celów. Mitchell stworzył jednak prosty przykład ilustrujący wiele koncepcji programowania genetycznego. Trzecie prawo ruchu planetarnego Keplera opisuje związek funkcjonalny między okresem orbitalnym planety P a jej średnią odległością A od Słońca. Funkcja trzeciego prawa Keplera, przy czym c jest stałą, to:

$P^2 = cA^3$

Jeśli przyjmiemy, że P jest wyrażone w latach ziemskich, a A w jednostkach średniej odległości Ziemi od Słońca, to c = 1. Wyrażenie s tej zależności to:

P = (sqrt (* A (* A A)))

Tak więc program, który chcemy rozwijać, jest reprezentowany przez strukturę drzewa na rysunku 7.



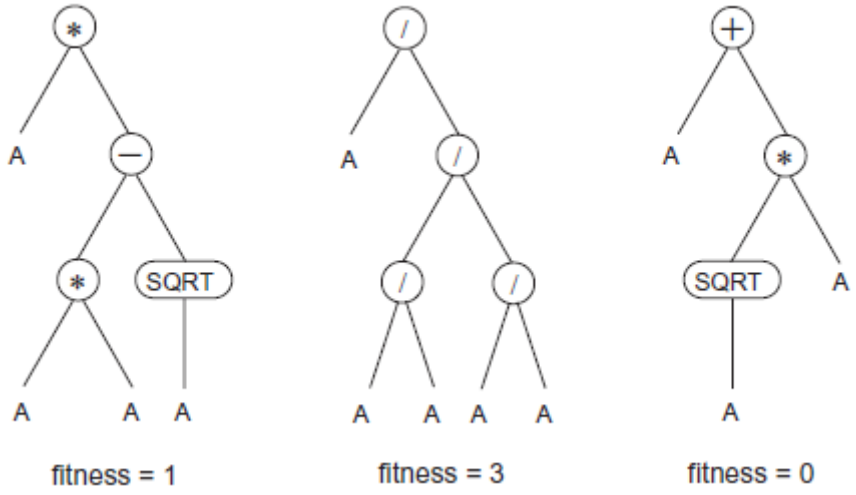
Wybór zestawu symboli terminala w tym przykładzie jest prosty; jest to pojedyncza wartość rzeczywista podana przez A. Zbiór funkcji mógłby być równie prosty, powiedzmy {+, -, *, /, sq, sqrt}. Następnie utworzymy początkową losową populację programów. Początkowa populacja może obejmować:

(* A (- (* A A) (sqrt A))) przydatność: 1

(/ A (/ (/ A A) (/ A A))) sprawność: 3

(+ A (* (sqrt A) A)) przydatność: 0

(Wkrótce wyjaśnimy załączone środki sprawności). Jak zauważono wcześniej w tej sekcji, ta inicjująca populacja często ma a priori ograniczenia zarówno rozmiaru, jak i głębokości wyszukiwania, biorąc pod uwagę wiedzę o problemie. Te trzy przykłady są opisane przez drzewa programów na rysunku 8



fitness = 1 fitness = 3 fitness = 0

Następnie określamy zestaw testów dla populacji programów. Załóżmy, że znamy pewne dane planetarne, które chcemy wyjaśnić w naszym wyewoluowanym programie. Na przykład, mamy dane planetarne w tabeli 3, zaczerpnięte z Urey, która zawiera zestaw punktów danych, które ewoluujące programy muszą wyjaśnić.

Planet	A (input)	P (output)
Venus	0.72	0.61
Earth	1.0	1.0
Mars	1.52	1.87
Jupiter	5.2	11.9
Saturn	9.53	29.4
Uranus	19.1	83.5

Ponieważ miara sprawności jest funkcją punktów danych, które chcemy wyjaśnić, definiujemy dopasowanie jako liczbę wyników programu, które mieszczą się w granicach 20% prawidłowych wartości wyjściowych. Używamy tej definicji do tworzenia miar sprawności trzech programów przedstawionych na rysunku 12.8. Czytelnikowi pozostaje stworzenie większej liczby członków tej początkowej populacji, zbudowanie operatorów krzyżowania i mutacji, które mogą tworzyć kolejne generacje programów, oraz określenie warunków zakończenia.

12.3 Sztuczne życie i uczenie się oparte na społeczeństwie

Wcześniej w tym rozdziale opisaliśmy uproszczoną wersję „Gry w życie”. Ta gra, najskuteczniej pokazana w komputerowych symulacjach wizualnych, w których kolejne generacje szybko się zmieniają i ewoluują na ekranie, ma bardzo prostą specyfikację. Po raz pierwszy została zaproponowana jako gra planszowa przez matematyka Johna Hortona Conwaya i rozsławiona dzięki omówieniu jej przez Martina Gardnera w Scientific American. Gra w życie to prosty przykład modelu obliczeniowego zwanego automatami komórkowymi (CA). Automaty komórkowe to rodziny prostych maszyn o skończonych stanach, które wykazują interesujące, wyłaniające się zachowania poprzez interakcje w populacji.

DEFINICJA

MASZYNA SKOŃCZONA (FSM) LUB AUTOMAT KOMÓRKOWY

1. Zestaw, który nazwałem alfabetem wejściowym.

2. Zbiór S stanów, w których może znajdować się automat.

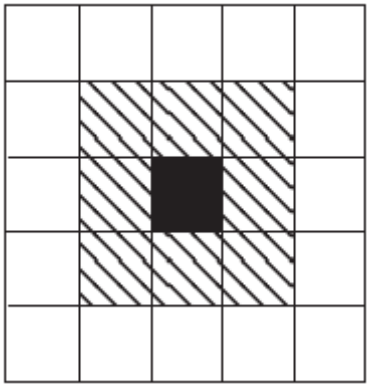
3. Wyznaczony stan s0, stan początkowy.

4. Następna funkcja stanu N: S x I → S, która przypisuje każdemu kolejny stan uporządkowaną parą składająca się z aktualnego stanu i wejścia prądowego.

Dane wyjściowe maszyny skończonej, jak przedstawiono wcześniej, są funkcją jej obecnego stanu i wartości wejściowych. Automaty komórkowe sprawiają, że wejście do obecnego stanu jest funkcją jego „sąsiednich" stanów. Zatem stan w czasie (t + 1) jest funkcją jego obecnego stanu i stanu jego sąsiadów w czasie t. To właśnie dzięki tym interakcjom z sąsiadami zbiory automatów komórkowych osiągają znacznie bogatsze zachowania niż proste maszyny skończone. Ponieważ wynik wszystkich stanów systemu jest funkcją ich sąsiednich stanów, możemy opisać ewolucję zbioru sąsiednich FSM jako adaptację i uczenie się oparte na społeczeństwie. W przypadku towarzystw opisanych w tym rozdziale nie ma jednoznacznej oceny sprawności poszczególnych członków. Sprawność jest wynikiem interakcji w populacji, interakcji, które mogą doprowadzić do „śmierci" poszczególnych automatów. Sprawność fizyczna jest nieodzowna w przetrwaniu jednostek z pokolenia na pokolenie. Uczenie się między automatami komórkowymi odbywa się zazwyczaj bez nadzoru; jak to ma miejsce w naturalnej ewolucji, adaptacja jest kształtowana przez działania innych, współewoluujących członków populacji. Globalny lub zorientowany na społeczeństwo punkt widzenia pozwala również spojrzeć na naukę z ważnej perspektywy. Nie musimy już skupiać się wyłącznie na jednostce, ale raczej dostrzegamy niezmienności i regularności pojawiające się w społeczeństwie jako całości. Jest to ważny aspekt badań Crutchfielda-Mitchella przedstawionych w sekcji 12.3.2. Wreszcie, w przeciwieństwie do nadzorowanego uczenia się, ewolucja nie musi być „zamierzona". Oznacza to, że społeczeństwo agentów nie musi być postrzegane jako „idące gdzieś", powiedzmy do jakiegoś punktu „omega". Mieliśmy tendencję do zbieżności, kiedy używaliśmy wyraźnych miar sprawności we wcześniejszych sekcjach tego rozdziału. Ale jak podkreśla Stephen Jay Gould, ewolucja nie musi być postrzegana jako „ulepszanie" rzeczy, a raczej sprzyja ona przetrwaniu. Jedynym sukcesem jest dalsze istnienie, a wzorce, które się wyłaniają, są wzorami społeczeństwa.

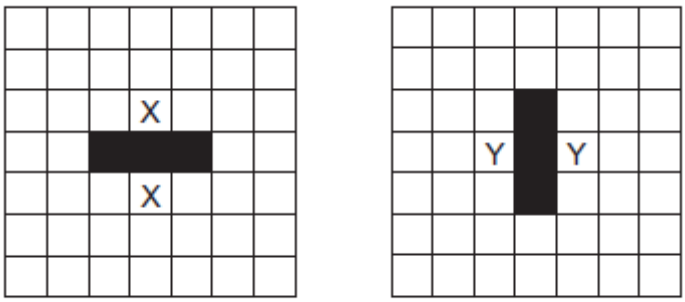
12.3.1 „Gra w życie"

Rozważmy prostą dwuwymiarową siatkę lub planszę do gry z rysunku 9.



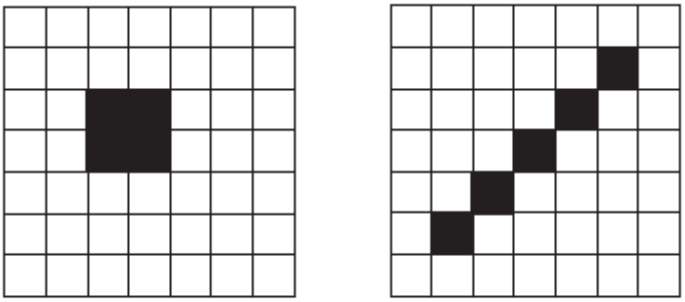
Tutaj mamy jeden kwadrat „zajęty" - o wartości bitu 1 - w kolorze czarnym, z ośmioma najbliższymi sąsiadami zaznaczonymi na szaro. Tablica jest przekształcana w okresach czasu, gdzie stan każdego kwadratu w czasie t + 1 jest funkcją jego stanu i stanu wskazanych najbliższych sąsiadów w czasie t.

Trzy proste zasady mogą napędzać ewolucję w grze: Po pierwsze, jeśli na jakimkolwiek polu, zajętym lub nie, są zajęte dokładnie trzy najbliższe sąsiadki, zostanie ono zajęte w następnym okresie. Po drugie, jeśli na jakimkolwiek zajętym polu są zajęte dokładnie dwóch najbliższych sąsiadów, zostanie ono zajęte w następnym okresie. Wreszcie we wszystkich innych sytuacjach plac nie będzie zajęty w następnym okresie. Jedną z interpretacji tych reguł jest to, że dla każdego pokolenia lub okresu życie w dowolnym miejscu, tj. To, czy kwadrat jest zajęty (ma wartość stanu 1), jest wynikiem życia własnego, jak również jego sąsiadów w okresie Poprzednia generacja. W szczególności zbyt gęsta populacja sąsiadujących sąsiadów (więcej niż trzech) lub zbyt mała populacja sąsiednich (mniej niż dwie) w dowolnym okresie nie pozwoli na życie następnemu pokoleniu. Rozważmy na przykład stan życia przedstawiony na rysunku 10a. Tutaj dokładnie dwa kwadraty, oznaczone x, mają dokładnie trzech zajętych sąsiadów. W następnym cyklu życia zostanie utworzony rysunek 10b. Tutaj znowu są dokładnie dwa kwadraty, wskazane przez y, z dokładnie trzema zajętymi sąsiadami. Można zauważyć, że stan świata będzie się zmieniać między rysunkami 10a i 10b.



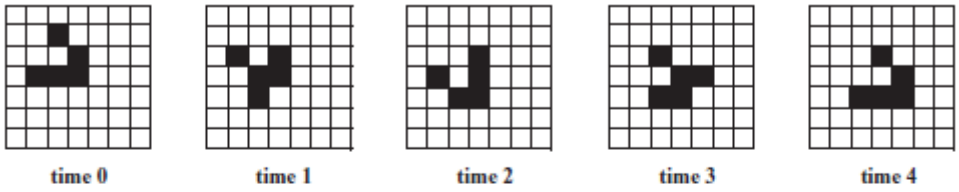
a. b.

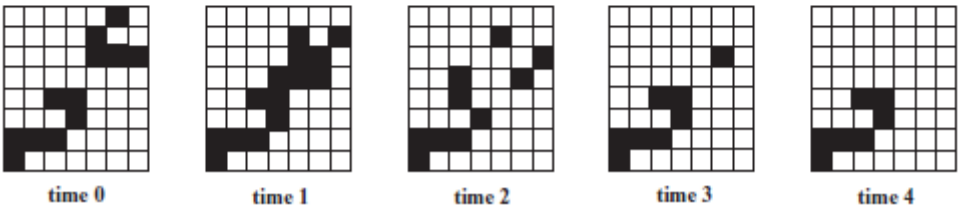
Czytelnik może określić, jaki będzie następny stan dla rysunków 11a i 11b oraz zbadać inne możliwe konfiguracje „świata”.



a. b.

Poundstone opisuje niezwykłą różnorodność i bogactwo struktur, które mogą pojawić się w grze życia, takich jak szybowce, wzory komórek poruszających się po świecie poprzez powtarzające się cykle zmian kształtu, jak pokazano na rysunku 12.





Ze względu na ich zdolność do tworzenia bogatych zachowań zbiorowych poprzez interakcje prostych komórek, automaty komórkowe okazały się potężnym narzędziem do badania matematyki wyłaniania się życia z prostych, nieożywionych składników. Życie sztuczne definiuje się jako życie stworzone raczej przez ludzki wysiłek niż przez naturę. Jak widać w poprzednim przykładzie, sztuczne życie ma silny smak „oddolny"; to znaczy, atomy systemu życia są definiowane i składane, a ich fizyczne interakcje „wyłaniają się". Regularności tej formy życia są uchwycone przez reguły skończonej maszyny stanowej. Ale jak można wykorzystać konstrukcje a-life? Na przykład w biologii zbiór żywych istot tworzonych przez naturę, tak złożony i różnorodny, jak tylko bywa, jest zdominowany przez przypadek i historyczną przygodność. Ufamy, że istnieją logiczne prawidłowości w tworzeniu tego zbioru, ale nie ma takiej potrzeby i jest mało prawdopodobne, że odkryjemy wiele z całkowitych możliwych prawidłowości, gdy ograniczymy nasz pogląd do zbioru bytów biologicznych, które faktycznie występują w naturze. zapewnia. Konieczne jest zbadanie pełnego zestawu możliwych prawidłowości biologicznych, z których część mogła zostać wyeliminowana przez przypadek historyczny. Zawsze możemy się zastanawiać, jak wyglądałby obecny świat, gdyby istnienie dinozaurów nie zostało definitywnie zakończone. Aby mieć teorię tego, co rzeczywiste, konieczne jest zrozumienie granic tego, co możliwe. Oprócz zdecydowanych wysiłków antropologów i innych naukowców, aby wypełnić luki w wiedzy o naszej rzeczywistej ewolucji, trwają spekulacje na temat ponownego przedstawienia historii samej ewolucji. Co by się stało, gdyby ewolucja rozpoczęła się w innych warunkach początkowych? A co by było, gdyby w naszym fizycznym i biologicznym otoczeniu wystąpiły alternatywne „wypadki" interwencji? Co może się pojawić? Co pozostanie niezmienne? Ścieżka ewolucyjna, która faktycznie miała miejsce na Ziemi, jest tylko jedną z wielu możliwych trajektorii. Na niektóre z tych pytań można by odpowiedzieć, gdybyśmy mogli wygenerować kilka z wielu biologiczne, które są możliwe. Technologia A-life nie jest tylko artefaktem dziedzin obliczeniowych lub biologicznych. Naukowcy z tak różnych dziedzin, jak chemia i farmakologia, zbudowali syntetyczne artefakty, wiele z nich związanych ze znajomością rzeczywistych bytów istniejących w naszym świecie. Na przykład w dziedzinie chemii badania nad budową materii i wieloma związkami dostarczanymi przez naturę doprowadziły do analizy tych związków, ich części składowych i wiązań. Ta analiza i rekombinacja doprowadziły do powstania wielu związków, które nie istnieją naturalnie. Nasza wiedza o elementach budulcowych przyrody doprowadziła nas do naszych własnych syntetycznych wersji, łączących elementy rzeczywistości w nowe i odmienne wzory. Dzięki tej dokładnej analizie naturalnych związków chemicznych dochodzimy do pewnego zrozumienia zestawu możliwych związków. Jednym z narzędzi do zrozumienia możliwych światów jest symulacja i analiza efektów ruchu i interakcji opartych na społeczeństwie. Mamy na to proste przykłady w Game of Life. Sekwencja cykli czasowych pokazana na rysunku 12 realizuje

wspomniany wcześniej szybowiec. Szybowiec przemierza przestrzeń gry, poruszając się po niewielkiej liczbie wzorów. Jego działanie jest proste, ponieważ w czterech przedziałach czasowych przesuwa się do nowej lokalizacji o jeden rząd dalej w prawo i jeden wiersz bliżej dolnej części siatki. Interesującym aspektem gry w życie jest to, że istoty takie jak szybowiec trwają do czasu interakcji z innymi członkami społeczeństwa; co się wtedy stanie, może być trudne do zrozumienia i przewidzenia. Na przykład na rysunku 13 widzimy sytuację, w której dwa szybowce wynurzają się i atakują. Po czterech okresach szybowiec poruszający się w dół i w lewo jest „konsumowany" przez drugą jednostkę. Warto zauważyć, że nasze opisy ontologiczne, to znaczy użycie przez nas terminów takich jak „byt", „migające światło", „szybowiec", „pochłonięty", odzwierciedla nasze własne antropocentryczne uprzedzenia w postrzeganiu form życia i interakcji, czy to sztucznych albo nie. Nadawanie nazw prawidłowościom pojawiającym się w naszych strukturach społecznych jest bardzo ludzkie.

12.3.2 Programowanie ewolucyjne

„Gra w życie" to intuicyjny, bardzo opisowy przykład automatów komórkowych. Możemy uogólnić naszą dyskusję o automatach komórkowych, charakteryzując je jako maszyny skończone. Omawiamy teraz społeczeństwa powiązanych FSM i analizujemy je jako powstające jednostki. To badanie jest czasami nazywane programowaniem ewolucyjnym. Historia programowania ewolucyjnego sięga początków samych komputerów. John von Neumann w serii wykładów w 1949 r. Zbadał, jaki poziom złożoności organizacyjnej jest wymagany do wystąpienia samoreplikacji (Burks 1970). Burks cytuje cel von Neumanna jako „… nie próbowanie symulowania autoreprodukcji naturalnego systemu na poziomie genetyki i biochemii. Chciał abstrahować od naturalnego problemu reprodukcji jego logicznej formy ". Usuwając szczegóły chemiczne, biologiczne i mechaniczne, von Neumann był w stanie przedstawić podstawowe wymagania dla samoreplikacji. von Neumann zaprojektował (jak nigdy dotąd) samoodtwarzający się automat składający się z dwuwymiarowego układu komórkowego zawierającego dużą liczbę indywidualnych 29-stanowych automatów, gdzie następny stan każdego automatu był funkcją jego obecnego stanu oraz stany jego czterech bezpośrednich sąsiadów. Co ciekawe, von Neumann zaprojektował swój samoreplikujący się automat, który, jak się szacuje, zawiera co najmniej 40000 komórek, miał funkcjonalność Uniwersalnej Maszyny Turinga. To uniwersalne urządzenie obliczeniowe miało również konstrukcję uniwersalną, w tym sensie, że było w stanie odczytać taśmę wejściową, zinterpretować dane na taśmie i za pomocą ramienia konstrukcyjnego zbudować konfigurację opisaną na taśmie w niezajętej części przestrzeń komórkowa. Umieszczając na taśmie opis samego automatu konstruującego, von Neumann stworzył samoodtwarzający się automat (Arbib 1966). Później Codd (1968) zredukował liczbę stanów wymaganych dla obliczeniowo uniwersalnego, samoodtwarzającego się automatu z 29 do 8, ale wymagał szacunkowo 100 000 000 komórek dla pełnego projektu. Później Devore uprościł maszynę Codda, aby zajmowała tylko około 87 500 komórek. W dzisiejszych czasach Langton stworzył samoreplikujący się automat, bez obliczeniowej uniwersalności, w którym każda komórka miała tylko osiem stanów i zajmowała tylko 100 komórek. Aktualne opisy tych wysiłków badawczych można znaleźć w materiałach z konferencji a-life. Tak więc formalna analiza samoreplikujących się maszyn ma głębokie korzenie w teorii obliczeń. Być może nawet bardziej ekscytujące wyniki są ukryte w badaniach empirycznych nad formami życia. O sukcesie tych programów nie świadczy jakaś funkcja sprawności a priori, ale raczej prosty fakt, że mogą one przetrwać i powielać się. Ich znakiem sukcesu jest przetrwanie. Po ciemnej stronie doświadczyliśmy spuścizny wirusów komputerowych i robaków, które są w stanie przedostać się do obcych hostów, replikować się (zwykle niszcząc wszelkie informacje w pamięci wymaganej do replikacji) i przenosić się do kolejnych obcych hostów. . Kończymy tę sekcję, omawiając kilka projektów badawczych, które można traktować jako przykłady obliczeń a-life. Ostatnia część rozdziału 12 to szczegółowy przykład wyłaniających się obliczeń Melanie Mitchell i jej współpracowników z Instytutu Sante Fe. Najpierw przedstawiamy dwa projekty omówione już w naszych wcześniejszych rozdziałach: Rodney Brooks z

MIT oraz Nils Nilsson i jego studenci ze Stanford. We wcześniejszych prezentacjach praca Brooksa znajdowała się pod ogólnym tytułem reprezentacji alternatywnych, a praca Nilssona, w temacie planowania. W kontekście teraźniejszości , przeformułowujemy ich prace w kontekście sztucznego życia i pojawiających się zjawisk. Rodney Brooks z MIT zbudował program badawczy oparty na założeniu a-life, a mianowicie, że inteligencja wyłania się poprzez interakcje wielu prostych autonomicznych agentów Brook opisuje swoje podejście jako „inteligencję bez reprezentacji", budując serię robotów zdolnych do wykrywania przeszkód i poruszania się po biurach i korytarzach MIT. Opierając się na architekturze subsumpcji, agenci mogą wędrować, badać i unikać innych obiektów. Inteligencja tego systemu jest artefaktem prostej organizacji i ucieleśnionych interakcji z otoczeniem. Brooks stwierdza: „Łączymy maszyny o skończonych stanach w warstwy kontroli. Każda warstwa jest zbudowana na wierzchu istniejących warstw. Warstwy niższego poziomu nigdy nie opierają się na istnieniu warstw wyższego poziomu ". Nils Nilsson i jego uczniowie opracowali program teleoreaktywny (TR) do kontroli agentów, program, który kieruje agenta w kierunku celu w sposób, który stale uwzględnia zmieniające się okoliczności środowiskowe . Ten program działa bardzo podobnie do systemu produkcyjnego, ale obsługuje również działanie trwające lub działanie, które ma miejsce w dowolnych okresach czasu, na przykład do ... Tak więc, w przeciwieństwie do zwykłych systemów produkcyjnych, warunki muszą być stale oceniane, a akcja związana z aktualnie najwyższym prawdziwym warunkiem jest zawsze wykonywana. W celu uzyskania dalszych szczegółów, zainteresowany czytelnik jest skierowany do Nilsson, Benson , Benson i Nilsson , Klein i inni. Komisja Europejska wspiera projekt PACE, Programmable Artificial Cell Evolution. Dzięki tym badaniom powstała nowa generacja syntetycznych ogniw chemicznych. Skupiamy się na tym, że życie ewoluuje wokół żywych systemów obliczeniowych, które same się organizują i samonaprawiają. Całe sztuczne komórki również rozmnażają się samoczynnie, są zdolne do ewolucji, mogą zachować swoją złożoną strukturę przy użyciu prostszych zasobów środowiskowych. Więcej szczegółów można znaleźć w witrynie internetowej PACE http://www.istpace.org/research_overview/artificial_cells_in_pace.html.

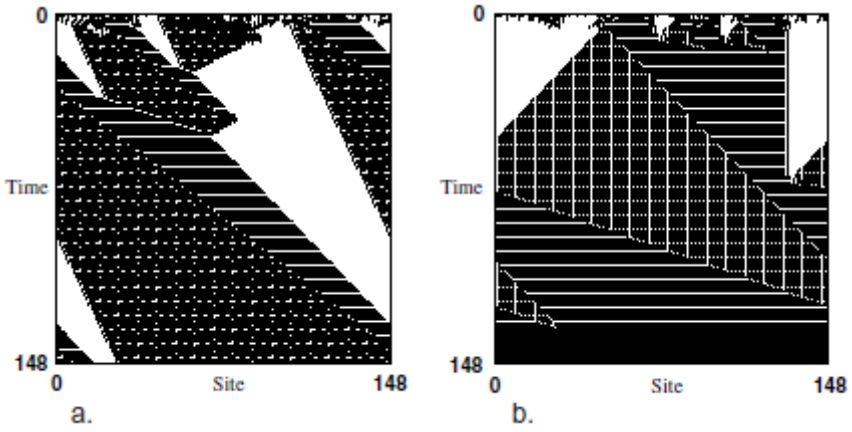
Wysiłki badawcze Johna Koza w zakresie programowania genetycznego na Uniwersytecie Stanforda nadal publikują wiele sukcesów. Ostatnie postępy obejmują automatyczny rozwój obwodów i sterowników, syntezę topologii obwodów, wymiarowanie, rozmieszczenie i routing, a także inne artefakty inżynieryjne. Odniesienia można znaleźć u Kozy.

Te cztery wysiłki badawcze są próbkami z bardzo dużej populacji projektów badawczych opartych na automatach. Te projekty są zasadniczo eksperymentalne. Zadają pytania dotyczące świata przyrody. Świat przyrody reaguje przetrwaniem i rozwojem udanych algorytmów, a także unicestwieniem systemu niezdolnego do adaptacji. Na koniec rozważymy badania przeprowadzone przez Santa Fe Institute: studium przypadku w wyłonieniu.

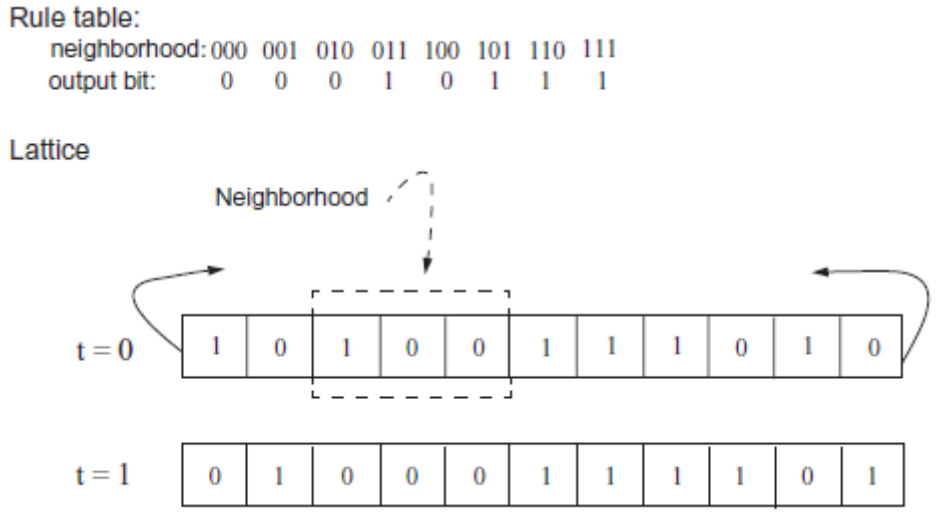
12.3.3 Studium przypadku w Emergence

Crutchfield i Mitchell badają zdolność ewolucji i interakcji w ramach prostych systemów do tworzenia relacji zbiorowego przetwarzania informacji wyższego poziomu. Ich badania stanowią przykład pojawienia się globalnych obliczeń w systemie przestrzennym składającym się z rozproszonych i oddziałujących lokalnie procesorów. Termin „emergent computation" opisuje pojawienie się globalnych struktur przetwarzania informacji w tych systemach. Celem badań Crutchfielda i Mitchella jest opisanie architektury i mechanizmów wystarczających do ewolucji i wspierania metod obliczeń wyłaniających się. Mówiąc konkretnie, automat komórkowy (CA) składa się z wielu pojedynczych komórek; w rzeczywistości w każdym automacie z przykładów, które prezentujemy, znajduje się 149 komórek. Te komórki stanu binarnego są rozmieszczone w jednowymiarowej przestrzeni bez globalnej koordynacji. Każda komórka zmienia stan w zależności od własnego stanu i stanów jej dwóch bezpośrednich sąsiadów. CA tworzy dwuwymiarową siatkę, gdy ewoluuje w różnych okresach czasu.

Sieć zaczyna się od początkowego losowo wygenerowanego zestawu N komórek. Na przykładzie z rysunku 12.14 istnieje 149 komórek i 149 etapów ich ewolucji. (Istnieje zerowy okres czasu, a komórki są ponumerowane od 0, 1, ..., 148). Dwa przykłady zachowania tych automatów komórkowych można zobaczyć na diagramach czasoprzestrzennych na rysunku 14.



Na tych diagramach jedynki są podane jako czarne pola, a 0 jako białe pola. Oczywiście różne reguły dotyczące sąsiedztwa komórek spowodują powstanie różnych wzorców na diagramie czasoprzestrzennym CA. Następnie opiszemy zestaw reguł, który określa aktywność komórek tworzących każdy ośrodek certyfikacji. Rysunek 15 przedstawia jednowymiarowy stan binarny najbliższego sąsiada CA, z N = 11 komórkami.



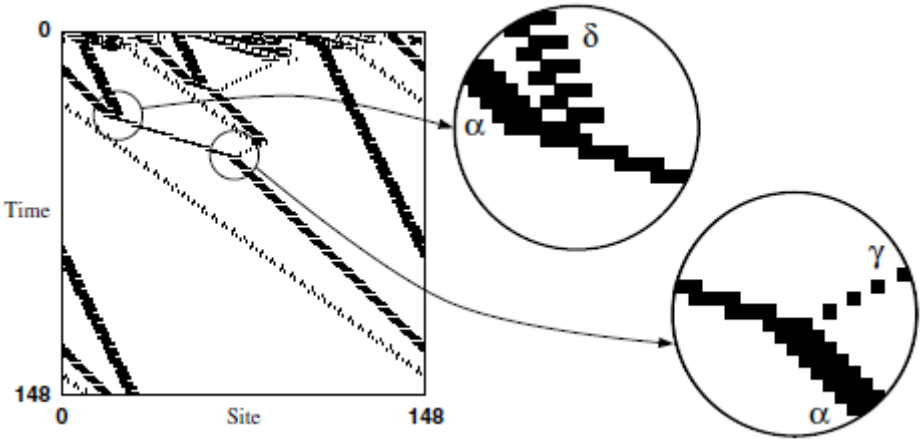
Przedstawiono zarówno siatkę, jak i tabelę reguł do aktualizacji sieci. Krata jest pokazana zmieniając się w jednym kroku czasowym. Krata jest właściwie walcem, z lewym i prawym końcem kraty w każdym przedziale czasowym sąsiadującymi (jest to ważne przy stosowaniu zestawu reguł). Tabela reguł obsługuje zasadę głosowania przez lokalną większość: jeśli lokalne sąsiedztwo trzech komórek ma większość jedynek, to w następnym kroku środkowa komórka staje się jedną; w przeciwnym razie staje się zerem w następnym kroku czasowym. Crutchfield i Mitchell chcą znaleźć ośrodek CA, który wykonuje następujące zbiorcze obliczenia, zwane tutaj większością wygrywa: jeśli początkowa sieć zawiera większość jedynek, CA powinien z czasem ewoluować do wszystkich; w przeciwnym razie powinna ewoluować do samych zer. Używają CA z sąsiedztwami zawierającymi siedem komórek, centralną komórkę z trzema sąsiadami po każdej stronie. Ciekawym aspektem tych badań jest to, że trudno jest zaprojektować regułę CA, która wykonuje obliczenia większości wygranych. W

rzeczywistości w Mitchell i inni wykazali, że prosta reguła „większości głosów” siedmiu sąsiadów nie wykonuje obliczenia „większość wygrywa”. GA służy do wyszukiwania reguły, która to zrobi. Algorytm genetyczny (GA), sekcja 12.1, służy do tworzenia tabel reguł dla różnych eksperymentów z CA. W szczególności GA służy do ewolucji reguł dla jednowymiarowej populacji komórek stanu binarnego, która tworzy każdy CA. Funkcja przystosowania ma na celu nagradzanie tych reguł, które wspierają wynik większościowy dla samego CA. Tak więc z biegiem czasu GA zbudowało zestaw reguł, którego przydatność była funkcją jej ostatecznego sukcesu w egzekwowaniu globalnych reguł większości. Najlepiej przystosowane reguły w populacji zostały wybrane, aby przetrwać i losowo łączone przez krzyżowanie w celu wytworzenia potomstwa, przy czym każde potomstwo podlega niewielkiemu prawdopodobieństwu mutacji. Proces ten był powtarzany przez 100 pokoleń, a przydatność szacowano dla nowego zestawu początkowych komórek w każdym pokoleniu. Pełne szczegóły można znaleźć w Crutchfield i Mitchell. W jaki sposób możemy określić ilościowo wyłaniające się obliczenia wspierane przez bardziej skuteczne CA? Podobnie jak wiele rozszerzonych przestrzennie procesów naturalnych, konfiguracje komórek często organizują się w czasie w regiony przestrzenne, które są dynamicznie jednorodne. W idealnym przypadku analiza i określenie podstawowych prawidłowości powinno być procesem zautomatyzowanym. W rzeczywistości Hanson i Crutchfield stworzyli język dla minimalnego deterministycznego automatu skończonego i używają go do opisu atraktorów-basenów w każdym automacie komórkowym. W tym języku można opisać nasz przykład. Czasami, jak na rysunku 14a, regiony te są oczywiste dla człowieka jako domeny niezmienne, to znaczy regiony, w których powtarza się ten sam wzór. Oznaczymy te domeny jako wartości Λ, a następnie odfiltrujemy niezmienne elementy, aby lepiej opisać interakcje (obliczenia), na które wpływają przecięcia tych domen. Tabela 4

Regular Domains		
$\Lambda^0 = 0^*$	$\Lambda^1 = 1^*$	$\Lambda^2 = (10001)^*$
Particles (Velocities)		
$\alpha \sim \Lambda^1\Lambda^0$ (1)	$\beta \sim \Lambda^0\Lambda^1$ (0)	
$\gamma \sim \Lambda^2\Lambda^0$ (–2)	$\delta \sim \Lambda^0\Lambda^2$ (1/2)	
$\eta \sim \Lambda^2\Lambda^1$ (4/3)	$\mu \sim \Lambda^1\Lambda^2$ (3)	
Interactions		
decay	$\alpha \rightarrow \gamma + \mu$	
react	$\alpha + \delta \rightarrow \mu,\ \eta + \alpha \rightarrow \gamma,\ \mu + \gamma \rightarrow \alpha$	
annihilate	$\eta + \mu \rightarrow \emptyset_1\ ,\ \gamma + \delta \rightarrow \emptyset_0$	

opisuje trzy regiony Λ: Λ0, powtarzające się 0; Λ1, powtarzające się jedynki; i Λ2, powtarzający się wzór 10001. Na rysunku 14a są inne regiony Λ, ale omówimy tylko ten podzbiór. Po odfiltrowaniu niezmiennych elementów domen Λ możemy zobaczyć interakcje między tymi domenami. W tabeli 4 opisujemy oddziaływanie sześciu obszarów Λ, na przykład cząstek na granicach domen Λ1 i Λ0. Granica, na której wszystkie domeny 1 spotyka się ze wszystkimi domenami 0, nazywana jest osadzoną cząstką α. Crutchfield i Mitchell twierdzi, że gromadzenie osadzonych cząstek jest głównym mechanizmem

przenoszenia informacji (lub sygnałów) w długich kontinuach czasoprzestrzennych. Logiczne operacje na tych cząstkach lub sygnałach zachodzą, gdy zderzają się. Zatem zbiór domen, ścian domenowych, cząstek i interakcji cząstek dla CA reprezentuje podstawowe elementy przetwarzania informacji, które są osadzone w zachowaniu CA, czyli składają się na wewnętrzne obliczenia CA. Jako przykład, rysunek 16 przedstawia logikę emergentną z rysunku 14a.



Obszary domeny Λ zostały odfiltrowane pod kątem ich niezmiennej zawartości, aby umożliwić łatwą obserwację cząstek ściany domeny. Każdy z powiększonych obszarów na rysunku 16 przedstawia logikę dwóch oddziałujących ze sobą osadzonych cząstek. Oddziaływanie cząstek $\alpha + \delta \rightarrow \mu$, pokazane w prawym górnym rogu, realizuje logikę odwzorowania konfiguracji przestrzennej reprezentującej sygnały α i δ na sygnał μ. Podobne szczegóły przedstawiono dla interakcji cząstek $\mu + \gamma \rightarrow \alpha$, która odwzorowuje konfigurację reprezentującą μ i γ na sygnał α. Bardziej kompletną listę interakcji cząstek z rysunku 12.16 można znaleźć w tabeli 4. Podsumowując, ważnym wynikiem badań Crutchfielda-Mitchella jest odkrycie metod opisywania nowych obliczeń w ramach przestrzennie rozproszonego systemu składającego się z lokalnie oddziałujących procesorów komórkowych. Lokalność komunikacji w komórkach narzuca ograniczenie globalnej komunikacji. Rolą GA jest odkrywanie lokalnych reguł komórkowe, których efektem jest przetwarzanie informacji na „duże" odległości czasoprzestrzenne. Crutchfield i Mitchell wykorzystali pomysły zaczerpnięte z formalnej teorii języka, aby scharakteryzować te wzorce czasoprzestrzenne. Dla Crutchfielda i Mitchella wynik ewoluującego automatu odzwierciedla całkowicie nowy poziom zachowania, różniący się od interakcji rozproszonych komórek na niższym poziomie. Globalne interakcje oparte na cząsteczkach pokazują, jak złożona koordynacja może wyłonić się w zbiorze prostych pojedynczych działań. Wynik GA działającego na lokalnych regułach komórkowych pokazał, jak proces ewolucyjny, wykorzystując pewne nieliniowe działania komórek tworzące wzorce, wytworzył nowy poziom zachowania i delikatną równowagę niezbędną do skutecznego wyłaniania się obliczeń. Wyniki badań Crutchfielda-Mitchella są ważne, ponieważ, przy wsparciu GA, wykazały pojawienie się niezmienności wyższego poziomu w świecie automatów komórkowych. Ponadto przedstawiają narzędzia obliczeniowe, zaadaptowane z formalnej teorii języka, które można wykorzystać do opisu tych niezmienności. Dalsze badania w tej dziedzinie mogą wyjaśnić pojawienie się złożoności: definiującej cechy żywych istot i fundamentalnej dla zrozumienia pochodzenia umysłów, gatunków i ekosystemów.