Algorytm Perkuna

Paweł Biernacki

pawel.f.biernacki@gmail.com

Vantaa 2021

wersja 0.0.2

Sztuczna Inteligencja jest Świętym Gralem współczesności. Istnieje moda na Sztuczną Inteligencję. Hasło to, wpisane po angielsku w popularną wyszukiwarkę daje 750 milionów rezultatów (sprawdzałem przed chwilą, 10.07.2021). Niestety ta popularność utrudnia przebicie się do publicznej świadomości takim jak ja, niszowym eksperymentatorom. Chciałbym w niniejszej książce opisać moją przygodę ze Sztuczną Inteligencją. Dokonałem pewnych odkryć, które chciałbym tu przedstawić w nadziei, że okażą się komuś pomocne. Pierwotnie nie zamierzałem pisać o tym książki. Miałem nadzieję przedstawić je w dyskusji z zainteresowanymi osobami. Niestety, wydaje się, że mimo współczesnej łatwości komunikacji nie będzie mi dane skontaktować się z takimi ludźmi. Próbowałem wiele razy, również na uniwersytetach. Generalnie nikt nie wierzy, że mogłem samodzielnie dokonać jakiegoś istotnego odkrycia. To trochę irytujące, ale z drugiej strony wbija mnie w dumę. Pamiętam pewnego profesora (w Niemczech), który powiedział mi: “Nie może Pan tak po prostu sobie czegoś napisać”. Inny profesor (w Finlandii) napisał mi, że nie będzie zainteresowany dopóki ta kwestia nie będzie ważna dla uniwersytetu. Odpisałem mu, że uniwersytet będzie niewątpliwie trwać, z moim algorytmem czy bez niego. I że ta kwestia jest istotna wyłącznie dla mnie. Pewien profesor z Rosji potraktował mnie wyjątkowo uprzejmie, zgodził się nawet żebym przedstawił studentom swoje osiągnięcia, ale zapewne zwątpił we mnie, bo kontakt się urwał. Wielu profesorów po prostu nie odpowiedziało na moją propozycję. Wysłałem (jeszcze jako student) opis mojego pomysłu na jakąś konferencję związaną ze Sztuczną Inteligencją. Został odrzucony. Nie miałem wtedy jeszcze implementacji. Pomyślałem, że napiszę w związku z tym książkę o charakterze popularno-naukowym, łatwiejszą do przebrnięcia niż suchy formalizm algorytmu lub kod źródłowy mojej implementacji. Książkę bez matematycznych wzorów ani nawet pseudokodu. Nawet bez diagramów.

Nie stosowałem w moich badaniach sieci neuronowych. Jak się zdaje większość współczesnych badaczy z nich korzysta. Nie wykluczam przejścia na sieci neuronowe w przyszłości, ale nie teraz. Ja lubię wiedzieć jak dany algorytm działa. Sieć neuronowa jeśli chodzi o jej architekturę jest pewną przestrzenią możliwych programów, a konkretne wartości jej parametrów są punktem w tej przestrzeni. Istota takiej “implementacji” pozostaje nieznana przed badaczem, zwłaszcza w przypadku dużych sieci. A ja lubię wiedzieć. Jak mawiał Dawid Hilbert - “Musimy wiedzieć, będziemy wiedzieć”. Zaprojektowałem mój algorytm od początku do końca i rozumiem jak on działa. Będę się starał tą wiedzą z Wami, drodzy Czytelnicy, podzielić.

Stworzyłem algorytm planowania dla pojedynczego agenta w stochastycznym świecie z niepełną informacją i dyskretnym czasem. Co to znaczy świat stochastyczny? To tak, jakby konsekwencje każdej mojej decyzji zależały w pewien sposób również czynnika stochastycznego, np. rzutu kośćmi. Inaczej niż np. w szachach czy w warcabach, w których świat jest deterministyczny. Można byłoby sobie wyobrazić jakiś rodzaj szachów, w których gracz podejmuje decyzję jedynie którą figurą lub pionkiem zamierza poruszyć, ale konkretne parametry tego ruchu (np. kierunek, odległość) są wybierane losowo, np. przez rzut kostki. Zatem ostatnie słowo co do konsekwencji ruchu należałoby do stochastycznego otoczenia. Co to znaczy świat z niepełną informacją? To znaczy, że pewne informacje o stanie świata są przed agentem ukryte, tak jak w pokerze. Nie wiemy jakie karty ma przeciwnik ani co jest na stosie. Przeciwnik wie, jakie karty ma, ale nie zna naszych kart i również nie wie, co jest na stosie. Nie stworzyłem algorytmu do planowania w grach z wieloma graczami. Mój algorytm radzi sobie dobrze tylko z jednym graczem. Nie znaczy to, że nie da się go stosować w grach z wieloma agentami. Po prostu pozostali agenci są przez mój algorytm traktowani jak część stochastycznego otoczenia. Nie wiemy nic o ich psychologii, o tym czego chcą, w co wierzą itd.

Założeniem mojego algorytmu jest planowanie zdobywania wiedzy. Jest to jego immanentna cecha, która mnie osobiście bardzo się podoba. Algorytm ten planuje przeprowadzanie eksperymentów (każda akcja jest eksperymentem) i optymalne wykorzystanie zdobytej w ten sposób wiedzy w celach strategicznych. Nie każda zmienna ukryta ma sens. Sens mają tylko takie zmienne ukryte, które w jakiś sposób wpływają na obserwowalną rzeczywistość. Nie zawsze sens ma płacenie w ten czy inny sposób za uzyskane dane. Czasem wystarcza takie czy inne założenie przyjęte na wiarę o stanie ukrytym, bez badania dokładnej wartości tego stanu. Przeprowadzenie

eksperymentu ma swój koszt i ma też potencjalny zysk, w tym sensie, że po nim strategiczna sytuacja agenta może się poprawić.

Wyszedłem z koncepcji zmiennych ukrytych. Doszedłem na chwilę obecną do pewnego algorytmu planowania/optymalizacji zaimplementowanego w języku i bibliotece o nazwie Svarog. Pierwotnie (kilka lat temu) zaimplementowałem nieco prostszą bibliotekę o nazwie Perkun, opartą na tym samym algorytmie. Mój algorytm radzi sobie dobrze w sytuacji, gdy mamy jednego inteligentnego agenta, ale nie wielu. Próbowałem rozwiązać ten problem dla wielu agentów i nie udało mi się to. Piszę o tym po to, żebyście mieli świadomość, drodzy Czytelnicy, jak długą drogę przeszedłem samotnie i jak dużo jeszcze przed nami.

Wyszedłem zatem z koncepcji zmiennych ukrytych. Rozpocząłem od prostych eksperymentów wykazujących, że jeżeli nie wiemy wszystkiego o świecie, to bardzo pomocne bywa wprowadzenie dodatkowych parametrów, które nie są bezpośrednio obserwowalne. Miewa to w pewnych grach dodatni wpływ na naszą zdolność do predykcji. To nie znaczy, że jakieś nieobserwowalne rzeczy istnieją “naprawdę” (co to w ogóle znaczy “naprawdę istnieć”), znaczy to tylko, że opłaca nam się czasem założyć, że istnieją. Oprócz zmiennych ukrytych mam, rzecz jasna, obserwowalne zmienne stanu, które nazywam zmiennymi wejściowymi. I mam też zmienne wyjściowe, których strumień wartości w czasie odpowiada wszystkim ludzkim (i nie tylko ludzkim) dziełom, wszystkim symfoniom, książkom, wierszom itd. Chodzi mi o strumień napięć całego zestawu mięśni podległych woli człowieka (z wyłączeniem np. mięśnia sercowego, bo on nie zależy od ludzkiej woli). Ten strumień napięć mięśni zawiera potencjalnie wszystko co człowiek w życiu może zrobić. W ciągu całego ludzkiego życia taki Beethoven czy Mozart nie mógł napisać, stworzyć niczego innego niż to co rozumiem przez strumień wartości zmiennych wyjściowych, od urodzin do śmierci. Gdy potraktujemy człowieka jako układ sterujący to widzimy w pierwszym przybliżeniu coś, co mapuje owe zmienne wejściowe na wyjściowe, w zależności od stanu jego mózgu. Zmiennych ukrytych nie widzimy. A jednak są one bardzo ważne, jak eksperymentalnie w kilku przypadkach (nie formalnie!) wykazałem. Człowiek jest czymś więcej niż funkcją mapującą z wejścia na wyjście. Nawiasem mówiąc jest też czymś więcej niż mój agent, ponieważ w moim algorytmie agent nie wyobraża sobie “stanów psychicznych” innego agenta. Nie jest w stanie tego zrobić. Aby uzyskać taki efekt potrzebny byłby jeszcze silniejszy algorytm od mojego.

Następnym krokiem było wprowadzenie funkcji wypłaty (ang. payoff function). Jest to funkcja w sensie matematycznym, czyli przyporządkowanie argumentom pewnych wartości. Liczb rzeczywistych. Stanowi ona co jest dla agenta dobre a co jest złe. W naturalny sposób określa ona silny porządek liniowy na zbiorze argumentów. Agent kierowany moim algorytmem stara się tak grać by zmaksymalizować wartość oczekiwaną tej funkcji w ciągu następnych n kroków gry. Pojawia się pytanie – czy ma ona zależeć wyłącznie od zmiennych wejściowych czy również od zmiennych ukrytych. Projektując algorytm podjąłem decyzję, by uzależnić ją wyłącznie od zmiennych wejściowych. Chciałem w ten sposób uniknąć sytuacji, w której agent wierzy w coś wyłącznie dlatego, że lubi w to wierzyć. Moi agenci wierzą jedynie w to, w co mają powód wierzyć. Ostateczną miarą sukcesu agenta jest wartość funkcji wypłaty dla strumienia danych wejściowych w czasie. Czyli strumień wypłaty.

Należy zaznaczyć, że mój agent może grać racjonalnie a pomimo to przegrać czyli doprowadzić się do jakiegoś niemożliwie beznadziejnego stanu. Maksymalizujemy wartość oczekiwaną, nadzieję matematyczną funkcji wypłaty. Grając racjonalnie można niestety wyjść na tym gorzej, czasami, niż korzystając z innych, mniej racjonalnych strategii. Kluczowe jest tu pojęcie głębokości drzewa gry, które nazywam horyzontem inteligencji. No i fakt, że mój agent **zazwyczaj** gra dobrze.

Chciałbym w tym miejscu wprowadzić rozróżnienie – co nazywam stanem a co stanem widocznym. Przez stan widoczny rozumiem wartości zmiennych wejściowych. Przez stan w ogóle rozumiem wartości wszystkich zmiennych stanu, czyli zarówno wejściowych jak i ukrytych. Każdemu stanowi widocznemu odpowiada pewien zbiór możliwych stanów, które są z nim zgodne co do wartości zmiennych wejściowych. Wiara, pojęcie, które wprowadzimy za chwilę, jest rozkładem prawdopodobieństwa nad tym zbiorem. Czyli najpierw mamy coś, co agent widzi,

następnie wyobraża sobie wszystkie możliwości na temat tego, czego nie widać (zmiennych ukrytych), a potem utrzymuje pewien rozkład prawdopodobieństwa nad tymi możliwościami, który nazywam wiarą.

Gdy wprowadzamy pewną ilość zmiennych ukrytych pojawia się niepewność, niewiedza co do ich rzeczywistych wartości. Nie będziemy tu wspominać mechaniki kwantowej i założymy jedynie, że każda zmienna ukryta ma w danej chwili konkretną wartość. Gdy ilość zmiennych ukrytych rośnie wówczas rośnie geometrycznie ilość możliwych stanów odpowiadających danemu stanowi widocznemu. Stan widoczny to wartości jedynie dla zmiennych wejściowych. Mój algorytm nie może zaprzeczyć temu co widzi, przeciwnie, to co widzi determinuje to w co agent może wierzyć. Aby manipulować wiedzą niepewną związaną ze zmiennymi ukrytymi wprowadzam pojęcie wiary (ang. belief). Ta wiara musi się zmieniać w odpowiedzi na reakcje otoczenia. Mamy pewną wiarę (opartą na przeszłości i tym co widzimy teraz), wykonujemy jakąś akcję i obserwujemy pewną reakcję otoczenia. Ta reakcja otoczenia prowadzi nas do nowej instancji naszej wiary, opartej o kolejną obserwację i znajomość tzw. modelu świata. Tak naprawdę w całych moich badaniach musiałem tylko dwa razy zrobić pewien skok, w pewnym sensie zaryzykować, nie dysponując twardymi dowodami. Był to moment, gdy opracowałem formułę w jaki sposób wiara zależy od poprzedniej wiary, ostatnio wykonanej akcji oraz reakcji otoczenia. Ta formuła jest dość prosta, ale nie była dla mnie oczywista. W chwili gdy ją miałem całość rozumowania była już tylko logiczną konsekwencją – mój agent konstruuje drzewo gry (tak jak w minimaksie, tylko pozbawionym członu “min”). W każdym węźle drzewa gry pojawia się nowa wiara – przekonanie na temat stanów zmiennych ukrytych. Drugim “skokiem” było znalezienie formuły, która zależnie od wiary, wykonanej akcji agenta i reakcji otoczenia ustali prawdopodobieństwo tej reakcji, czyli nowego stanu widocznego. Gdy mam drzewo gry z prawdopodobieństwami reakcji otoczenia dla założonych akcji agenta wówczas mogę obliczyć wartości funkcji wypłaty dla reakcji otoczenia i ich wartości oczekiwane. Wykonując mój algorytm agent “myśli” następująco: mam teraz to co widzę oraz pewną wiarę. Jeżeli wykonam jedną akcję to mogę dojść do wachlarza rezultatów z danymi (zależnymi od mojej wiary) prawdopodobieństwami, obliczyć dla nich wartość oczekiwaną funkcji wypłaty, a jeżeli wykonam inną akcję wówczas mogę dojść do innego wachlarza rezultatów z danymi prawdopodobieństwami. W każdym kolejnym kroku planowania wyobrażam sobie kolejne wiary, czyli interpretacje zaobserwowanej hipotetycznie reakcji otoczenia.

To co napisałem powyżej można zrekapitulować następująco – mój agent potrafi wyobrazić sobie z jakim prawdopodobieństwem **otrzyma** dany rezultat, czyli reakcję otoczenia (stan widoczny), oraz potrafi wyobrazić sobie w jaki sposób **zinterpretuje** ten rezultat, czyli w co będzie wierzył jeśli go rzeczywiście otrzyma. Wyobrażenie sobie tych dwóch rzeczy jest kluczowe. Aby sobie to wyobrazić potrzebna jest informacja o poprzedniej wierze, wykonanej ostatnio akcji i zakładanym rezultacie, czyli stanie widocznym.

Mamy więc drzewo gry z obliczonymi w każdym węźle wiarami i możemy wybrać taką akcję, by wartość oczekiwana funkcji wypłaty była maksymalna. Gdy planujemy na jeden krok do przodu jest to trywialne. Wybieramy po prostu tą akcję, dla której wartość oczekiwana funkcji wypłaty w następnym kroku jest największa. Dla wielu kroków musimy założyć, że po pierwszym kroku nadal będziemy postępowali optymalnie, czyli będziemy wybierać akcje wynikające z naszego algorytmu. Jest to więc algorytm rekurencyjny, co nie dziwi, zważywszy, że drzewa gry i w ogóle drzewa są strukturami rekurencyjnymi.

Nasz agent propaguje w przyszłość swoje interpretacje hipotetycznych rezultatów eksperymentów, czyli swoje wiary. Każda akcja jest jednocześnie eksperymentem, pytaniem zadanym otoczeniu. Wyobrażając sobie rezultaty eksperymentów i dokonując ich interpretacji agent planuje aż do n kroków na przyszłość. Nie ma w tym algorytmie miejsca dla innych agentów. To jest słabość mojego algorytmu. Inni agenci muszą być z konieczności traktowani tak, jakby byli częścią stochastycznego środowiska. Nie mamy wglądu w ich “psychikę”, nie wiemy co myślą o tym, co ja wiem, co myślą o tym co ja myślę, że oni myślą itd.

Po znalezieniu odpowiednich formuł udało mi się napisać implementację, która zachowywała się sensownie. Pierwszą taką implementacją był Perkun. W Perkunie model świata

jest przedstawiany w sposób dość prosty, za pomocą prawdopodobieństw przejścia z jednego stanu do drugiego, przy założonej akcji. Następną i, jak na razie, ostatnią implementacją mojego algorytmu jest Svarog. Zmienne ukryte, wejściowe i wyjściowe są w nim wprowadzane tak jak w Perkunie, ale model świata i funkcja wypłaty są już wyrażane za pomocą dość silnego języka reguł. Logicznie każdy program w Svarogu ma swój odpowiednik w Perkunie, tylko ze względu na olbrzymie rozmiary modeli nie zawsze jest sens po Perkuna sięgać. Ogólnie Perkun jest lepszy dla prostszych przypadków, ponieważ jest bardziej intuicyjny. Svarog jest za to silniejszy. Aby zniwelować problemy z Perkunem gdy nie istniał jeszcze Svarog opracowałem pewien sposób generowania jego specyfikacji przy użyciu Prologu. Kto ciekaw ten może zajrzeć do projektu PerkunWars na Sourceforge, katalog perkun.

Obie biblioteki (Perkun i Svarog) zostały napisane w C++, można je wykorzystywać we własnych programach C++. Istnieją do nich interpretery, np. program perkun jest interpreterem języka Perkun, a program svarog interpreterem języka Svarog. Wprowadzanie danych w linii poleceń do interpretera jest dość toporne i polecam to wyłącznie w przypadku testowania. Przy wykorzystaniu we własnych programach proponuję stworzenie odrębnego procesu dla każdego interpretera i komunikowanie się z tym procesem przez tzw. potoki (ang. pipes). Istnieją dwa przykładowe programy wykorzystujące tą technikę - wspomniany już PerkunWars dla Perkuna oraz Dorban dla Svaroga. Oba te programy są dostępne na Sourceforge.

Moja implementacja (mówię o Svarogu) zachowywała się poprawnie, ale była nieznośnie powolna. W przykładowym demku napisanym dla Svaroga – w Dorbanie planowanie dla czterech ruchów naprzód mogło trwać pół godziny i dłużej. Wówczas zacząłem poszukiwać sposobu zoptymalizowania Svaroga. I znalazłem ją. Kluczem do sukcesu okazała się dyskretyzacja zbioru możliwych wiar, a następnie zapisanie tzw. prekalkulowanej wiedzy do specyfikacji Svaroga. Prekalkulowana wiedza ma szereg zalet. Po pierwsze jej rozmiar nie rośnie wraz ze wzrostem ilości kroków planowania naprzód. Po drugie pozwala ona na podjęcie decyzji (wybór optymalnej akcji) w czasie rzędu dwustu-trzystu milisekund. Po napisaniu specyfikacji Svaroga można zapuścić komputer do prekalkulowania wiedzy (w przypadku Dorbana było to pięć dni, na dwunastu logicznych procesorach). Jest więc tak, że najpierw piszemy specyfikację Svaroga, testujemy ją, a następnie prekalkulujemy zachowanie agenta dla wszystkich możliwych sytuacji (stanów wejściowych oraz wiar). Z przestrzenią wiar jest problem, ponieważ jest to przestrzeń ciągła, stąd przy prekalkulacji Svarog dokonuje jej dyskretyzacji. Nawiasem mówiąc gdybyśmy nie brali pod uwagę wiary wówczas utknęlibyśmy w martwym punkcie, co, jak się zdaje, przytrafiło się wielu badaczom Sztucznej Inteligencji. Problem ten nie jest oczywisty. Nawet dla szachów nie jest możliwe zapamiętanie wszystkich możliwych stanów wejściowych, czyli stanów na planszy. Ten fakt przysłania nieco konieczność uwzględnienia jeszcze zmiennych ukrytych, bo wprawdzie w szachach ich nie ma, ale już w pokerze czy brydżu są. Zmienne ukryte powodują konieczność zarządzania wiarą czyli wiedzą o tym, co niewidoczne. Musimy sobie wyobrazić co byśmy zrobili gdybyśmy widzieli X a w dodatku wierzyli w Y. Ominięcie tego drugiego członu prowadzi do impasu. W Dorbanie jasno widać, że tytułowy bohater – właśnie Dorban – w identycznym stanie widocznym potrafi robić różne rzeczy – zależnie od tego w co akurat wierzy. Pomimo to algorytm sterujący Dorbanem jest całkowicie deterministyczny. Nie ma w nim nic przypadkowego.

Mała dygresja. A propos prekalkulacji naszej wiedzy o grze, w którą wszyscy gramy. Być może do tego właśnie służą ludziom marzenia oraz wyrzuty sumienia – do obliczania sobie optymalnych rozwiązań w różnych sytuacjach. Prawdziwy inteligentny agent nie będzie potrzebował uczyć się całe życie jak człowiek – wystarczy, że damy mu, powiedzmy, rok lub dwa na zastanowienie się nad zadanym modelem świata i już po roku/dwóch tego zastanawiania się będzie od razu zachowywał się inteligentnie. Być może Sztuczna Inteligencja będzie w przyszłości wymagała nie tyle szybkich komputerów co pojemnych dysków. Nasuwa się pewna analogia – wiedza prekalkulowana wygląda dla agenta trochę jak “religia” bo zawiera informacje o optymalnej w danej sytuacji akcji bez uzasadnienia dlaczego tak jest. Prawdziwe religie opierają się zazwyczaj na postulowanym przez nie Absolucie, ale w rzeczywistości są, jak sądzę, funkcjonalnym odpowiednikiem mojej prekalkulowanej wiedzy. Nawiasem mówiąc dyskretyzacja przestrzeni wiar

nie jest panaceum, bo często zdarza się, że przestrzeń ta “eksploduje” ze względu na zbyt dużą liczbę wymiarów. W takim wypadku Svarog zaznacza sobie w specyfikacji, że dany przypadek jest zbyt złożony i robi obliczenia wyłącznie dla tzw. wektorów bazowych. To jak wykorzystuję te wektory bazowe przekracza ramy tej książki, ale nie jest to matematycznie trudne.

W Dorbanie mamy graf złożony z pięciu miast, oraz trzy postacie: Pregora (kontrolowanego przez człowieka), Dorbana (kontrolowanego przez Svaroga) oraz wampira. Dorban wie tylko w jakim mieście jest, czy widzi Pregora, czy widzi Pregora żywego, czy widzi wampira oraz czy Pregor za nim podąża. Generalnie Dorban nie wie, gdzie jest wampir, gdzie jest Pregor i czy Pregor jest żywy. Są to dla niego zmienne ukryte. W tej grze pewne stany są niemożliwe – np. nie można widzieć wampira jeżeli jest on w innym mieście. Jeżeli Dorban widzi wampira, to musi on być w tym samym mieście co Dorban, czyli niemożliwe jest, by wampir był gdzie indziej. Słowo wyjaśnienia odnośnie tej informacji czy Dorban widzi Pregora żywego – Pregor może umrzeć i nawet zmartwychwstać (ang. resurrect). Jeżeli Dorban nie widzi Pregora wówczas ta zmienna - “widzę Pregora żywego” ma wartość “żaden” - ang. none (czyli ani prawda ani fałsz). Pomimo to Dorban może pamiętać (dzięki mechanizmowi wiary), że Pregor ostatni raz przezeń widziany był np. martwy i wówczas nie będzie się spodziewał jego pomocy. To nie znaczy, że Pregor NAPRAWDĘ jest teraz martwy, bo mógł wszak zmartwychwstać będąc w innym mieście, ale Dorban nie bierze tej możliwości pod uwagę. Dorban pamięta stan Pregora kiedy go ostatnio widział, więc gdy raz ujrzy go martwego przestanie zabiegać o jego pomoc i samotnie ruszy na poszukiwanie wampira. Ciekawe jest jego zachowanie, jeżeli Pregor (żywy) będzie odmawiał współpracy. Wówczas Dorban będzie w kółko prosił go o współpracę. Dzieje się tak dlatego, że odpowiedź Pregora jest zaimplementowana jako zdarzenie stochastyczne, które w kolejnych próbach ma pewne prawdopodobieństwo pozytywnego rezultatu. Dorban nie “uczy się” na przykład z historii, że proszenie o pomoc Pregora wydaje się daremne.

Proponuję uruchomić sobie Dorbana i kilkakrotnie odpalić opcję “nic nie rób” (ang. do nothing). To polecenie “nic nie rób” odnosi się do Pregora. Dorban-bohater zacznie szukać wampira i Pregora, a gdy spotka tego ostatniego poprosi go o pomoc. W momencie, w którym prosi o pomoc może jeszcze nie wiedzieć gdzie jest wampir, ale z drzewa gry wynika, że tu czy gdzie indziej wampira w końcu znajdzie i wtedy pomoc Pregora mu się przyda. Gdy Pregor podąża za Dorbanem wówczas powinien używać opcji menu “wykonuj rozkazy” (ang. follow orders). Chyba, że chce odejść i grać na własną rękę.

Od czasu do czasu może się w Dorbanie zdarzyć sytuacja, którą nazywam niespodzianką (ang. surprise). Jest tak na przykład wtedy, gdy Pregor gdzieś sobie pójdzie (albo nagle przyjdzie do Dorbana), albo na jego oczach umrze czy zmartwychwstanie. Taka sytuacja powoduje reset wiary, to znaczy wszystko co nie jest niemożliwe (w świetle tego co Dorban widzi) staje się równie prawdopodobne. To może mało finezyjny sposób obsługi takiej sytuacji, ale wydał mi się najprostszy. Każdy może obsłużyć niespodziankę na własny sposób, redefiniując w swoim kodzie odpowiednią wirtualną funkcję w klasie dziedziczonej z klasy svarog::optimizer. Jeżeli niespodzianki nie obsłużymy wówczas nasz program przerwie interakcję i rzuci wyjątek.

Moim marzeniem byłoby stworzenie czegoś podobnego do Svaroga dla wielu agentów, być może z ciągłym (a nie dyskretnym) czasem i możliwością wprowadzania zmiennych ukrytych dynamicznie. Może z czymś w rodzaju teorii klas i obiektów, jak w programowaniu obiektowym?

Zatem, reasumując, mamy agenta, który utrzymuje pewną wiarę opartą o swoją historię. Wiara ta reprezentuje wiedzę dotyczącą zmiennych ukrytych i wynika z jego historii. Konieczne są dwa wzory – do znalezienia prawdopodobieństwa pewnej reakcji otoczenia oraz do znalezienia interpretacji pewnej reakcji otoczenia (ta interpretacja jest nową wiarą). Ten wzór do znalezienia interpretacji jest wykorzystywany nie tylko w planowaniu, ale i w pętli agenta. Po każdej obserwacji następującej po wykonaniu jakiejś akcji obserwacja ta jest interpretowana – tj. tworzy się nowa instancja wiary. Miałem określoną wiarę warunkowaną przez to co widziałem oraz moją historię, wykonałem akcję i zaobserwowałem pewną reakcję otoczenia. Konstruuję w związku z tym interpretację – nową wiarę. I tak każdym kroku. To bardzo istotne dla algorytmu, że wzór na interpretację obserwacji jest użyty w dwóch miejscach. Sztuka interpretacji jest nam potrzebna

dwukrotnie – raz gdy planujemy i interpretujemy sobie to co może się wydarzyć po naszych akcjach, oraz drugi raz, gdy rzeczywiście coś się wydarzy.

Mój algorytm, jak już wspominałem, planuje zdobywanie wiedzy. Na bieżąco ocenia czy opłaca się poprzestać na wiedzy niepewnej (wierze) czy też lepiej ją uściślić przez wykonanie jakiegoś eksperymentu. To nie znaczy, że zawsze dążymy do zdobycia maksymalnej możliwej wiedzy. Taka strategia byłaby nierozsądna. Raczej staramy się do zdobycia wiedzy tam, gdzie jej koszt jest niższy niż strategiczny zysk. Rezultatem eksperymentu, czyli wykonania jakiejś akcji, może być jedna z wielu możliwych reakcji otoczenia, które odpowiadają stanom widocznym. Dorban na przykład planuje, że znajdzie Pregora, poprosi go o pomoc, a następnie obaj poszukają razem wampira (chyba, że Dorban już wie, gdzie jest wampir) i wspólnie zaatakują go. Jeżeli Pregor nie żyje wówczas Dorban będzie szukał wampira na własną rękę, aby go zaatakować. W programie tym jak dotąd (wersja Dorbana 0.0.2) nie ma możliwości, żeby zabić wampira, ale Dorban o tym nie wie. Zresztą nie bierze pod uwagę śmierci wampira, po prostu lubi z nim wygrywać. Mimo, że istnieje pięć miast Dorban musi odwiedzić maksymalnie cztery z nich, żeby uzyskać pewność na temat tego gdzie jest np. wampir. Jeżeli w żadnym z dotychczas odwiedzonych czterech miast wampira nie ma, wówczas musi on być w pozostałym – piątym mieście. Jest to możliwe dzięki zastosowaniu mojego wzoru na interpretację hipotetycznej reakcji otoczenia. A co jeśli wampir sam z siebie się ruszy w trakcie gdy przemierzamy owe cztery miasta? Może się to skończyć niespodzianką – tak przynajmniej jest to rozwiązane w Dorbanie.

Opiszę teraz nieco bardziej szczegółowo samo drzewo gry. W drzewie gry są dwa rodzaje węzłów i dwa rodzaje krawędzi. W korzeniu drzewa mamy **węzeł wiary**. Z tego węzła (któremu odpowiada pewien stan widoczny oraz wiara) rozchodzą się krawędzie, które odpowiadają wyborowi agenta, czyli akcjom. Nazwijmy je **krawędziami akcji**. Po każdej takiej krawędzi następuje węzeł, który nazwalibyśmy **węzłem stochastycznej reakcji otoczenia**. Po węźle stochastycznej reakcji otoczenia mamy drugi rodzaj krawędzi – **krawędź stochastycznej reakcji otoczenia**. Takie krawędzie nie zależą już od woli agenta, lecz mają przypisane pewne prawdopodobieństwo. Każda taka krawędź kończy się znowu węzłem wiary (któremu odpowiada pewien stan widoczny oraz wiara). Może to być liść, a może być i korzeń poddrzewa. W każdym węźle wiary mamy pewien stan widoczny a zatem możemy obliczyć wartość funkcji wypłaty. W każdym węźle stochastycznej reakcji otoczenia możemy obliczyć wartość oczekiwaną wykorzystując prawdopodobieństwa z krawędzi reakcji otoczenia oraz wartości funkcji wypłaty z węzłów wiary. W węźle wiary wybieramy tą akcję, która daje maksymalną wartość oczekiwaną. Tak to wygląda dla drzewa o wysokości 2. Co z drzewem o wysokości 4 i więcej? W każdym węźle wiary za wyjątkiem liści dodajemy do funkcji wypłaty maksimum z wartości oczekiwanej uzyskanej dla akcji i dopiero tę wartość przekazujemy do obliczenia wartości oczekiwanej w węźle stochastycznej reakcji. Ten algorytm można by nazwać “stochmaks” przez analogię do minimaksa. Nie ma w nim przeciwnika, inaczej niż w minimaksie. Jest tylko jeden agent “walczący” ze stochastycznym otoczeniem,

Należy zaznaczyć, że nawet uważna lektura niniejszej książki nie pozwoli jeszcze na rekonstrukcję algorytmu. Brakuje w niej owych dwóch ważnych wzorów – jak interpretować reakcję otoczenia (czyli jak tworzyć nową wiarę) oraz jak obliczać prawdopodobieństwo reakcji otoczenia. Tego już nie da się powiedzieć inaczej niż językiem matematyki. Ewentualnie zalecam analizę kodu źródłowego Svaroga (klasa svarog::optimizer). Ale będę zadowolony jeżeli uda mi się zainteresować Was, drodzy Czytelnicy, samą ideą.

Chciałbym jeszcze dodać – czysto hipotetycznie – jak widzę przyszłość mojego algorytmu. Powinien stać się podstawą do opracowania kolejnych algorytmów, takich, w których agenci mają świadomość istnienia innych agentów, być może każdy z własną funkcją wypłaty. Jeszcze lepiej – gdy agent A myśli, że agent B ma daną funkcję wypłaty. Również modele świata powinny się w nich znajdować z klauzulą: A myśli, że B myśli, że ... . Chodzi nie tylko o samą wiarę. Również o model. W takich algorytmach w naturalny sposób powinna się pojawić możliwość komunikacji między agentami. W moim algorytmie model świata jest stały i z góry zadany. Agent nie uczy się modelu świata, tylko działa w ramach jednego, narzuconego z góry modelu. To pewne

ograniczenie. Co to w ogóle jest model świata? Jest to prawdopodobieństwo przejścia w jednym kroku (czas jest dyskretny) z jednego stanu do drugiego stanu, przy założeniu że wykonujemy daną akcję. Przez “stan” rozumiem wartości wszystkich zmiennych stanu, czyli zarówno zmiennych wejściowych jak i zmiennych ukrytych. W Perkunie ten model świata jest dosłownie czymś takim. W Svarogu jest on mniej widoczny, gdyż jest zapisywany w formie specjalnych reguł, ale każdy program w Svarogu ma swój perkunowy ekwiwalent. Do modelu należą też informacje o stanach niemożliwych. Na przykład jeśli Dorban jest w Krakowie i widzi wampira, to niemożliwe jest, żeby wampir nie był w Krakowie. To nie jest jakaś uniwersalna własność, bo potrafię sobie wyobrazić grę, w której Dorban może widzieć wampira choć jest on w innym mieście. Zarówno Perkun jak i Svarog wspierają definiowanie pewnych stanów jako “niemożliwych”. Jest to, jak mi się zdaje, jakiś odpowiednik logiki. Nie idzie o to, by z zadanych aksjomatów wywnioskować jakieś twierdzenia. Rzecz jest bardziej pierwotna, po prostu pewne stany są niemożliwe.

W Perkunie i w Svarogu istnieje też możliwość zadeklarowania, że w pewnej sytuacji (chodzi o stan widoczny) użycie danej akcji jest zabronione. Nielegalne. Na przykład jeśli Dorban jest w Krakowie, to nie ma sensu wykonanie akcji “idź do Krakowa”. Dlaczego zależy to jedynie od stanu widocznego, a nie od stanu generalnie? Chodzi o to, aby nie było wątpliwości, że akcja jest nielegalna, niezależnie od wiary, czyli opinii agenta na temat wartości zmiennych ukrytych. Decyzję o tym, czy akcja jest legalna czy nie można podjąć wiedząc jedynie co agent widzi.

Perkun i Svarog stanowią implementacje nieco nietypowego sposobu programowania. Zamiast mówić komputerowi co ma robić, jak to ma miejsce w paradygmacie imperatywnym, informujemy go co może zrobić (jakie są legalne akcje), co lubi a czego nie lubi (funkcja wypłaty) oraz jakie są konsekwencje jego decyzji (model świata). Svarog (ale nie Perkun) potrafi dodatkowo przeanalizować tak przedstawione informacje i przetworzyć je do wiedzy prekalkulowanej. Krok ten nie wnosi nic nowego do samej istoty algorytmu, z wiedzą prekalkulowaną czy bez niej rezultat, czyli podejmowane decyzje, powinnien być ten sam. Wiedza prekalkulowana po prostu powoduje, że proces liczenia jest znacznie szybszy. Stanowi pewnego rodzaju pamięć podręczną (ang. cache). Wiedzę prekalkulowaną możemy w Svarogu sprokurować dopiero wtedy, gdy wiemy, że nasz model świata jest poprawny. Oznacza to, że w każdej sytuacji prawdopodobieństwa przejścia sumują się do jedynki dla każdej akcji. Svarog wspiera sprawdzanie modelu świata za pomocą specjalnej instrukcji “test”. Instrukcja ta wypisuje na standardowy strumień błędu wszelkie odchylenia od normy naszego modelu świata, z pewną założoną tolerancją.

Myślę, że w przyszłości będzie można kupić “modele świata” i narzucać komputerowi jedynie funkcję wypłaty, tj, mówić mu czego po jego zachowaniu oczekujemy. Programista przyszłości mówiłby komputerowi jedynie czego od niego chce, ale nie informowałby go bezpośrednio w jaki sposób to osiągnąć. Należy zaznaczyć, że mój model świata w Perkunie i w Svarogu jest dość silnie zintegrowany. Nie znajduję lepszego słowa. Nie ma w nim np. pojęcia klasy takiego jak w programowaniu obiektowym. Nie ma pojęcia obiektów. Każda zmienna (ukryta bądź wejściowa) jest jedyna w swoim rodzaju. Unikalna. Zmienne ukryte są wprowadzone raz na zawsze, są z góry zadane. Nie jest więc możliwe, by agent wprowadzał sobie zmienne ukryte dynamicznie, np. pod wpływem jakichś odkryć. W moim algorytmie nie jest to możliwe, ale generalnie, w jakimś innym – kto wie.

Wiedza prekalkulowana może być opracowana w Svarogu za pomocą instrukcji precalculate. Jeśli użyjemy jej prostej wersji wówczas obliczenia będą przeprowadzane tylko na jednym procesorze. Jeżeli mamy tak jak ja, na jednej maszynie dwanaście lub więcej logicznych procesorów wówczas opłaca się maksymalnie zrównoleglić obliczenia. Służą do tego perlowe skrypty, których nazwy zaczynają się od “svarog_generate”. Po szczegóły odsyłam Cię, drogi Czytelniku, do dokumentacji Svaroga. Skrypty te korzystają ze specjalnej wersji instrukcji precalculate – a mianowicie takiej, która działa tylko dla określonego stanu widocznego. Korzystają też z narzędzia (dostarczanego razem ze Svarogiem), które nazywa się svarog_merge. Jest to program pozwalający na połączenie wiedz prekalkulowanych z dwóch plików. Dzięki temu możemy maksymalnie wykorzystać możliwości naszej maszyny, przeprowadzając obliczenia (wiedzy prekalkulowanej) na wszystkich procesorach jednocześnie.

Normalnie korzystamy z Perkuna lub Svaroga w trybie interaktywnym. Podajemy jakie dane są obserwowane na wejściu (wartości zmiennych wejściowych) a następnie Perkun/Svarog podejmuje decyzję co do zmiennych wyjściowych. Jeżeli uruchomimy interpreter z poprawną specyfikacją z terminala, wówczas będziemy mogli obserwować "wiarę" naszego agenta – jak się ta wiara zmienia przy przechodzeniu przez kolejne sytuacje. Krok po kroku wiara jest uaktualniana, po pierwsze dlatego, że zależy od stanu widocznego, czyli informacji na wejściu, po drugie dlatego, że zależy od poprzedniej wiary, ostatnio wykonanej akcji i otrzymanego rezultatu. Jedynym wyjątkiem jest pierwsza wiara, która przybiera formę wiary a priori. Wiara a priori zakłada wszystkie możliwe stany z równym prawdopodobieństwem. Jest to pewne ograniczenie, przyznaję. Perkun lub Svarog nieustannie interpretują kolejne rezultaty (stany widoczne) i nieustannie dopasowują swoją wiarę do otrzymywanych rezultatów, starając się odgadnąć "prawdziwe" wartości zmiennych ukrytych. Piszę "prawdziwe" w cudzysłowie, dlatego że agent nie ma możliwości sprawdzić jakie są poprawne odpowiedzi na sformułowane przezeń pytania, które określilibyśmy jako semantykę zmiennych ukrytych. Agent jedynie zakłada, że zmienne ukryte "naprawdę istnieją". Zakłada to z powodów czysto pragmatycznych.

Jeżeli mamy program taki jak PerkunWars albo Dorban, który korzysta z Perkuna/Svaroga jako z biblioteki wówczas sprawa przedstawia się nieco inaczej. Powinniśmy stworzyć klasę dziedziczoną z odpowiedniego optymalizatora i powiedzieć komputerowi, redefiniując odpowiednie funkcje wirtualne, w jaki sposób nasz optymalizator pobiera dane wejściowe oraz w jaki sposób wykonuje optymalną akcję. Zazwyczaj interaktywna pętla interpretera będzie działała w odrębnym procesie, komunikując się przez potoki z głównym procesem. Główny proces przesyła do procesu-dziecka informacje o stanie widocznym, czyli wartości zmiennych wejściowych i otrzymuje od niego wybraną optymalną akcję. Poziom "inteligencji" (głębokość drzewa gry) w tego typu programach jest stały i niezmienny. Na przykład w Dorbanie jest to 4. Można ten parametr zwiększyć lub zmniejszyć zmieniając odpowiednio wartość parametru instrukcji loop w danej specyfkacji Svaroga. Należy zauważyć, że zmniejszenie tego parametru zasadniczo nic nie da, ponieważ Svarog i tak będzie korzystał z prekalkulowanej wiedzy, która jest obliczona dla wartości 4. Natomiast jego zwiększenie spowoduje, że część obliczeń będzie musiała być dokonana w czasie rzeczywistym, przez co czas obliczenia się zwiększy. Jednak dla wartości 5 i więcej Dorban będzie "bardziej inteligentny", to jest będzie pracował z większą głębokością drzewa gry.

W Dorbanie główny bohater może poprosić Pregora o pomoc i robi to natychmiast, gdy go spotka (są wyjątki od tej reguły, ale o tym za chwilę). Robi to nawet wtedy, gdy nie wie jeszcze gdzie jest wampir. Po prostu wampir gdzieś być musi i z drzewa gry wynika, że prędzej czy później go znajdziemy. Wyjątki od wspomnianej reguły to na przykład taka sytuacja, gdy wampir jest w Gdańsku i Dorban o tym wie, a Pregor jest we Wrocławiu. Policzmy ilość potrzebnych kroków. Prośba o pomoc Pregora we Wrocławiu to jeden, ruch Dorbana i Pregora do Krakowa to dwa, ruch Dorbana i Pregora do Warszawy to trzy, ruch Dorbana i Pregora do Gdańska to cztery, zaatakowanie wampira to pięć. Pięć ruchów! Ale wysokość drzewa gry jest ustalona na cztery! Dlatego Dorban "nie jest w stanie wyobrazić sobie" żeby poproszenie Pregora o pomoc we Wrocławiu dało mu jakąś korzyść, ponieważ jego "horyzont inteligencji" jest zbyt krótki. Zbyt ograniczający na to, by dostrzec korzyści z tej decyzji. Dopiero gdybyśmy ustawili poziom inteligencji na 5 wówczas Dorban zauważyłby, że nawet i w tej sytuacji opłaca się poprosić Pregora o pomoc.

Dlaczego Dorban w ogóle prosi Pregora o pomoc? Czy nie mógłby atakować wampira sam? Oczywiście mógłby i gdy Pregor umrze (na przykład w konsekwencji nieudanego ataku na wampira – to się zdarza) wówczas Dorban istotnie zacznie atakować wampira na własną rękę. Prosi go o pomoc wyłącznie dlatego, że z modelu świata wie, że jego szanse przeciwko wampirowi wzrosną jeżeli Pregor będzie mu pomagał. Dorban lubi wygrywać z wampirem i nie lubi z nim przegrywać. Jedna ze zmiennych wejściowych nazywa się "czy_Dorban_wygrał_walkę" i zazwyczaj jest równa "żaden" (ang. none), ale po walce przybiera wartość "prawda" lub "fałsz". Dorban dlatego poszukuje wampira i stara się go atakować, o ile możności z asystencją Pregora, że wartość funkcji

wypłaty dla tej zmiennej wejściowej (czy_Dorban_wygrał_walkę) jest wyższa dla prawdy niż dla fałszu.

Razem ze Svarogiem dostarczane są programy svarog-daemon oraz svarog-dummy-client. Svarog-daemon jest, jak sama nazwa wskazuje, demonem. Słucha on na danej maszynie poleceń, które przybierają formę kompletnych specyfikacji Svaroga. Specyfikacje takie nie mogą zawierać komendy loop (pętli), ponieważ nie działają w sposób interaktywny. Mogą one wyłącznie rozwiązywać jakiś jednostkowy problem, tj. znajdować wartość zadanej akcji dla określonej wiary i stanu widocznego. Należy zaznaczyć, że to rozwiązanie (svarog-daemon) jest przestarzałe. Powstało ono w czasach, gdy Svarog nie potrafił jeszcze tworzyć prekalkulowanej wiedzy. Obecnie bardziej opłaca się wykorzystać prekalkulowaną wiedzę niż korzystać ze svarog-demonów. Tak też robi Dorban. Można go skonfigurować tak, by korzystał nawet z kilku maszyn z uruchomionymi demonami, niemniej jednak będzie to działało wolniej niż bez tej konfiguracji – po prostu dzięki prekalkulowanej wiedzy. Wspominam o tym rozwiązaniu jedynie dla porządku.

W najnowszej wersji Dorbana (0.0.2) istnieje możliwość “teleportowania” wampira do dowolnego, losowo wybranego miasta. Jeżeli wampir nagle zniknie, a Dorban go widział, wówczas Dorban będzie, oczywiście, zaskoczony. Jego model świata nic nie mówi o teleportacjach wampira. Mógłby oczywiście mówić, ale nie mówi. Przypominam, że owo zaskoczenie ma konkretne znaczenie, wówczas Dorban zresetuje swoją wiarę tak, by wszystko co jest możliwe (w świetle tego co widzi) było równie prawdopodobne. Na przykład zaskoczony Dorban jeżeli nie widzi Pregora przestanie zakładać, że Pregor nie żyje nawet jeżeli myślał tak poprzednio i założy z równym prawdopodobieństwem że Pregor żyje jak i że nie żyje. Jego wiara zostanie niejako zastąpiona przez wiarę a-priori, uwarunkowaną jedynie tym co widzi. Ten nieco dziwny efekt to konsekwencja wybranej przeze mnie metody obsługi “zaskoczenia”, ale ktoś inny tworząc podobny do Dorbana własny projekt oparty na Svarogu mógłby obsłużyć to w inny sposób.

W Svarogu model świata jest reprezentowany przez hierarchiczne zagnieżdżone struktury akcja/przypadek/prawdopodobieństwo. Dla każdej zmiennej np. wejściowej lub ukrytej możemy odnosić się do jej wartości pierwotnej (ang. initial value - przed wykonaniem akcji) oraz wartości końcowej (ang. terminal value – po wykonaniu akcji). Dodatkowo możemy formułować własności modelu wykorzystując funkcje (funkcje nie w sensie matematycznym, a raczej programistycznym). Są to proste procedury obliczające i zwracające jakąś wartość. Można w nich stosować instrukcję warunkową oraz instrukcję zwracającą wartość (ang. return). Nie ma pętli, zmiennych lokalnych czy złożonych struktur danych. Cały model świata jest zawarty w strukturze o nazwie “wiedza” (ang. knowledge). Gdy zobaczymy specyfikację Svaroga używaną przez Dorbana zauważymy w niej również wiedzę prekalkulowaną. Są to reguły stworzone automatycznie, na podstawie analizy poprawnego i przetestowanego modelu świata. Nie należy próbować pisać takich reguł w wiedzy prekalkulowanej ręcznie!

Oprócz struktur akcja/przypadek/prawdopodobieństwo wiedza w Svarogu może też zawierać informacje o tym kiedy stan jest niemożliwy oraz o funkcji wypłaty. Może być wiele takich informacji. Jeżeli warunki w jakiejkolwiek sekcji “niemożliwe” (ang. impossible) są spełnione, wówczas dany stan jest niemożliwy.

Jeżeli znasz Perla, drogi Czytelniku, zachęcam Cię byś rzucił okiem na plik svarog/perl/dorban_svarog_file_generator.pm w rozpakowanym Dorbanie. To tu powstaje specyfikacja Svaroga dla Dorbana. Listę miast oraz istniejące między nimi połączenia zawiera plik svarog/perl/list_of_towns.pm. Plik svarog/perl/parameters.pm zawiera wartości prawdopodobieństw oraz wartości funkcji wypłaty, które zobaczymy później w specyfikacji. Nawiasem mówiąc plik svarog/perl/list_of_towns.pm ma wpływ tylko na zachowanie Dorbana i służy tylko do utworzenia specyfikacji Svaroga, więc graf miast użyty w programie C++ jest niejako zduplikowany jeszcze w samym programie. Moglibyśmy poinformować Dorbana, że graf wygląda inaczej, np. że jest bezpośrednie połączenie z Krakowa do Gdańska i wówczas Dorban zachowywałby się inaczej! Nawiasem mówiąc poruszanie się po krawędziach tego grafu Pregora nie obowiązuje – może on skoczyć do dowolnego miasta (wystarczy na nie kliknąć). Ostateczną specyfikacją Svaroga używaną w Dorbanie jest plik svarog/dorban.svarog. Zawiera on już prekalkulowaną wiedzę, przez

co jest spory (ok. 2MB), ale za to działa bardzo szybko. Jeżeli Cię to interesuje możesz, drogi Czytelniku, poszukać w tym pliku sekwencji tekstu “too complex”. To są właśnie przypadki, gdy Svarog w trakcie prekalkulacji zdecydował, że dyskretyzacja przestrzeni wiar nie ma sensu ze względu na ilość wymiarów (możliwych stanów) i przeprowadza obliczenia, jak już wspomniałem, jedynie dla wektorów bazowych.

W Dorbanie jeżeli zgodzimy się (jako Pregor) na towarzyszenie Dorbanowi (bohaterowi), wówczas powinniśmy używać opcji menu “wykonuj rozkazy” (ang. follow orders). Wówczas Dorban zacznie nas informować o tym, co mamy robić a my możemy jego propozycję przyjąć bądź odrzucić. Odrzucenie propozycji lub w ogóle wybranie innej opcji menu niż “wykonuj rozkazy” oznacza, że Pregor opuszcza Dorbana. Jeżeli Dorban nie wie jeszcze gdzie jest wampir, wówczas będzie chciał, żebyśmy go razem z nim szukali. Jeżeli już wie – zaprowadzi nas do niego najkrótszą drogą. Ciekawe jest to, że jeśli np. spotkał wampira w Krakowie to nie zaatakuje go od razu, tylko sprawdzi Wrocław i Warszawę – czy nie ma tam przypadkiem Pregora. Akurat w tym wypadku inteligencja 4 wystarcza na wykonanie (w “myśli”) sekwencji – idź do Warszawy, poproś o pomoc, wróć do Krakowa, zaatakuj wampira, bo są to cztery ruchy. Podobnie jest z sekwencją – idź do Wrocławia (z Krakowa), poproś o pomoc, wróć do Krakowa, zaatakuj wampira, bo są to również cztery ruchy. Gdyby Dorban myślał, że Pregor jest w Poznaniu lub w Gdańsku, to i tak nie wpadnie na to, by się tam udać, gdyż jego horyzont inteligencji na to nie zezwala (wymagałoby to większej ilości ruchów niż 4).

Wyobrażam sobie, że jeszcze lepszy od mojego algorytm byłby w stanie założyć zmienność modelu. Modele same z siebie są olbrzymie, więc ich przestrzeń musi być jeszcze większa. Nie obyłoby się bez jakiejś sprytnej optymalizacji. Wówczas mielibyśmy być może agenta zdolnego do “trenowania” swojego modelu gry, ale postępującego zgodnie z rygorystycznym algorytmem optymalizacji. Mam na myśli jakąś hybrydę mojego algorytmu z sieciami neuronowymi, ale w tym miejscu powściągnę wodze fantazji.

Napisałem, że inni gracze są przez mojego agenta traktowani jako część stochastycznego otoczenia. Co to znaczy? Na przykład Dorban wie o tym, że Pregor istnieje, ale traktuje go, a raczej jego zgodę na współpracę, jako pewną zmienną losową. Nie znudzi mu się proszenie Pregora o pomoc tak długo aż tamten się zgodzi, albo ... umrze. Ta ostatnia możliwość (śmierć Pregora na oczach Dorbana, bez żadnej walki) spowodowałaby zaskoczenie tego ostatniego. Z drugiej strony może się zdarzyć, że Pregor zginie w walce z wampirem. Szanse na to nie są duże, ale są. Jeśli Dorban to zobaczy wówczas nie będzie zaskoczony. Jego model świata dopuszcza taką ewentualność. Natomiast nie dopuszcza zmartwychwstania Pregora, choć jest ono możliwe. Stąd zacznie atakować wampira samodzielnie.

Chciałbym jeszcze napisać o pewnej optymalizacji, którą uważny programista dostrzeże w moim kodzie. Każdemu stanowi widocznemu odpowiada cały zbiór możliwych stanów, rozkład nad którym jest wiarą mojego agenta. Otóż ten zbiór stanów jest generowany dopiero wtedy, gdy jest potrzebny. Zaraz po tym jest niszczony. Do tego zbioru stanów odnoszą się wiary, które są wszak rozkładami prawdopodobieństwa nad nim. Taka wiara nie może istnieć bez istnienia stanów z tego zbioru. Wynika stąd, że i wiara jest niszczona natychmiast gdy przestaje być potrzebna. Przechodząc przez drzewo gry rozważamy sekwencyjnie wszystkie ścieżki od korzenia do kolejnych liści i wówczas, dla każdego ze stanów widocznych znajdujących się na danej ścieżce, musimy mieć te zbiory stanów “rozwinięte”. Może się zdarzyć, że na ścieżce od korzenia do pewnego liścia ten sam stan widoczny występuje więcej niż jeden raz. Wówczas wystarczy, by rozwinąć odpowiedni zbiór stanów tylko raz (utrzymuję pewien rodzaj liczników). Przez tę optymalizację program jest nieco wolniejszy, gdy nie ma prekalkulowanej wiedzy, ale za to zezwala na większą ilość zmiennych ukrytych. Starsze wersje Svaroga, pozbawione tej optymalizacji, generowały zbiory stanów dla wszystkich możliwych stanów widocznych zaraz po uruchomieniu.

Gdybyśmy stworzyli specyfikację Svaroga z prawdopodobieństwami równymi 1.0, wówczas nasz świat byłby w pełni deterministyczny. Jest to więc przypadek ogólniejszy, działający również dla światów deterministycznych. Gdybyśmy **nie dali** żadnych zmiennych ukrytych, wówczas wiara zdegenerowałaby się do rozkładu nad jednoelementowym zbiorem. Podobnie jest z

Perkunem. Wówczas dla każdego stanu widocznego wierzymy w to samo i w ogóle uproszczona jest interpretacja kolejnych stanów widocznych. Można też tworzyć zmienne ukryte, które z założenia nie mają wpływu na rezultat żadnej akcji. Takie zmienne są jednak zbędne. To pewna egzemplifikacja słynnej brzytwy Ockhama.

W tak prostych programach jak Dorban wystarczy jedna zmienna wyjściowa, której możliwe wartości to "idź do ...", albo "zaatakuj wampira" albo "poproś Pregora o pomoc". Perkun i Svarog wspierają jednak użycie wielu zmiennych wyjściowych. Te dodatkowe zmienne mogą być "argumentami" dla głównej zmiennej. Ich sens tkwi w modelu świata należącym do wiedzy w Svarogu lub w modelu świata w Perkunie.

Wyobrażam sobie następcę mojego algorytmu jako algorytm, w którym wszyscy agenci podejmują decyzje jednocześnie i każdy z nich dysponuje w ogólnym wypadku pewną wiedzą o innych agentach. Na przykład agent A myśli, że agent B myśli, że agent C lubi X. To odnosi się oczywiście do funkcji wypłaty. Ale i może być tak, że agent A myśli, że agent B myśli, że agent C wierzy w X. To odnosi się do wiary. Podobnie powinno być z modelami świata. Racjonalna strategia w grze z nieracjonalnymi agentami może być bardzo interesująca.

W bardzo wielu znanych mi językach programowania istnieje instrukcja warunkowa. Jej wykonanie polega na ewaluacji pewnego warunku sformułowanego dla pewnych zmiennych i, jeżeli warunek ten jest spełniony, na wykonaniu pewnego bloku instrukcji. Myślę, że dotykamy tu bardzo pierwotnej koncepcji Sztucznej Inteligencji. Jeżeli A to wykonaj B. Wszystkie zmienne w znanych mi językach programowania (poza Perkunem i Svarogiem) są zmiennymi widocznymi. Jest to do tego stopnia nagminne, że muszę wyjaśniać po co potrzebuję zmiennych ukrytych i jak je stosować. Można oczywiście argumentować, że warunek A może obejmować zarówno informacje bezpośrednio postrzegalne jak i to, co nazywam wiarą. Ale w praktyce tak nie jest. W praktyce w naszym myśleniu o Sztucznej Inteligencji wpadliśmy w pułapkę instrukcji warunkowej i chcielibyśmy uzależnić nasze akcje wyłącznie od tego, co widzimy, nie od tego, w co wierzymy. W świetle mojego doświadczenia zdobytego przy tworzeniu Perkuna i Svaroga jest to błąd. Zmienne ukryte naprawdę mają dużo do zaoferowania, przede wszystkim dlatego, że pozwalają na pewną kumulację wiedzy wynikającej z historii agenta. Wyobraźmy sobie, że Dorban jest w Krakowie i widzi tam wampira. Nie atakuje go jednak – idzie do Warszawy. Załóżmy, że w Warszawie nie ma Pregora. Dorban wraca do Krakowa, znowu widzi tam wampira i tym razem idzie do Wrocławia. Stan widoczny jest ten sam co na początku, znowu jest w Krakowie i znowu widzi wampira, ale w pierwszym kroku w takiej sytuacji idzie do Warszawy, a w drugim do Wrocławia. Wydaje się, jakby algorytm był niedeterministyczny, ale tak nie jest. Algorytm jest deterministyczny. Po prostu Dorban wie już, że w Warszawie Pregora nie ma. Był w Warszawie i pamięta, że wszystkie stany zawierające informację "Pregor jest w Warszawie" musiał wykluczyć. Dlatego gdy ponownie jest w Krakowie i widzi to samo (wampira), to jednak sytuacja jest inna, gdyż **wierzy** w co innego. Poprzednio dopuszczał możliwość, że Pregor jest w Warszawie, teraz, ze względu na swoją historię, już jej nie dopuszcza. Dlatego za drugim pobytem w Krakowie idzie do Wrocławia. Dlatego też właśnie wiedza prekalkulowana składa się z sekcji "w stanie widocznym"/"w wierze" (ang. on visible state, on belief), czyli zależy od obydwu, od stanu widocznego oraz od wiary agenta. To, że w językach programowania nie ma na ogół pojęcia zmiennej ukrytej, czyli zmiennej o nieznanej wartości powoduje, że umyka nam konieczność uwzględnienia wiary w naszych regułach. Popełniliśmy pewien grzech – nie uwzględniliśmy wiary i to się na nas zemściło. Gdybyśmy uwzględnili wiarę, to nawet bez algorytmu Perkuna moglibyśmy dojść chociażby do pewnej jego emanacji jaką jest wiedza prekalkulowana. Problem ze zmiennymi ukrytymi jest taki, że przy naiwnej (takiej jak moja) implementacji przestrzeń możliwych stanów "eksploduje". Załóżmy, że mamy 10 zmiennych binarnych, czyli mających wartości "prawda" lub "fałsz". Ilość możliwych stanów momentalnie rośnie do 1024 możliwości. A 10 zmiennych to bardzo mało. W językach programowania (innych niż Perkun i Svarog) trzeba się porządnie napisać, żeby stworzyć pojęcie stanu, cóż dopiero mówić o zbiorze możliwych stanów i o rozkładzie prawdopodobieństwa nad tym zbiorem, który nazywam wiarą. Normalne języki programowania nie wspierają tej koncepcji, bo jest to koncepcja nowa. Mówię o koncepcji zmiennych ukrytych. W normalnych językach

programowania każda zmienna ma jakąś tam wartość, może niepoprawną, ale ma. Brak zmiennych ukrytych wspieranych przez normalne języki programowania powoduje niestety, że brakuje nam intuicji w jaki sposób ich używać. Na przykład w Dorbanie zmienną ukrytą jest "gdzie_jest_wampir". Jeżeli wampira widzimy to z wiedzy o stanach niemożliwych możemy wydedukować, że jest on tam gdzie Dorban. Jeżeli go nie widzimy, to wiemy jedynie, że musi być gdzie indziej. Ale to nie wszystko – powiedzmy, że jesteśmy w Krakowie. Nie widzimy wampira. Wierzymy zatem, że wampir może być w Warszawie, Poznaniu, Wrocławiu lub w Gdańsku. A co, jeżeli w Warszawie już byliśmy? I nie widzieliśmy tam wampira? Wówczas będziemy wierzyć, że wampir może być jedynie w Poznaniu, Wrocławiu lub w Gdańsku. Widzimy, że nasza historia wpływa na to, w co wierzymy. To, w co wierzymy jest dla podjęcia decyzji jeszcze bardziej istotne niż to, co widzimy. Albo przynajmniej równie istotne.

Zmienne (ukryte i inne) w Perkunie i Svarogu przypominają nieco tzw. typy wyliczeniowe, jeśli chodzi o możliwe wartości. Nie ma zmiennych typu całkowitego czy rzeczywistego. Oczywiście Perkun i Svarog są jedynie pewnego rodzaju zabawkami, mającymi stanowić demonstrację pewnego konceptu. Algorytm powinien działać również dla zmiennych typu całkowitego i rzeczywistego. Ilość możliwych stanów rośnie wówczas do nieskończoności i potrzebujemy jakiejś sprytnej optymalizacji, by operować na wierze jako rozkładzie prawdopodobieństwa nad nieskończonym zbiorem. Jestem optymistą – myślę, że to jedynie kwestia czasu. Myślę, że mamy w zasięgu ręki algorytmy stosujące praktycznie matematyczne metody obliczania jakichś zmiennych ukrytych, coś w rodzaju kwadratur, które stosowano do przybliżonego całkowania, gdy nie istniały jeszcze komputery. Te kwadratury to jedynie przykład, chodzi mi o systematyczne stosowanie matematyki. Aktualnie komputery stosują matematykę wtedy, gdy każemy im to robić. Ale w przyszłości – kto wie, być może będą potrafiły decydować samodzielnie o tym gdzie, kiedy i jak należy ją stosować. Aby to nastąpiło, muszą najpierw dowiedzieć się, że czegoś nie wiedzą. Muszą poznać koncepcję zmiennych ukrytych.

Popuszczając nieco wodze fantazji – chciałbym zobaczyć naukę albo inżynierię tworzoną przez Sztuczną Inteligencję. Kiedyś zrobiłem taki eksperyment – w Perkunie (Svarog jeszcze wtedy nie istniał). Napisałem program, który tworzy programy w pewnym prymitywnym języku. Każdy taki tworzony program miał dwie komendy, bez żadnych parametrów. Ten prymitywny język dopuszczał jedynie dwie instrukcje. Jak łatwo policzyć daje to cztery możliwe programy. Zmienne wejściowe zawierały program, była też jedna dodatkowa zmienna wejściowa mówiąca, czy dany program działa. Jak łatwo się domyślić Perkun "lubił", kiedy ta ostatnia zmienna wejściowa była równa "prawdzie". Były też cztery zmienne ukryte, każda mówiąca, czy odpowiadający jej program działa. Zmienne wyjściowe sterowały prostymi operacjami na programie, tak, by można było w skończonej liczbie kroków przejść wszystkie możliwości. Dodatkowo była operacja "wykonaj program" czy też raczej "sprawdź czy program działa". I to właśnie Perkun robił – przechodził przez wszystkie możliwe programy pytając czy dany program działa. Należy zaznaczyć, że nie dysponował żadną wiedzą odnośnie semantyki tych programów, żadnym doświadczeniem jakim dysponuje prawdziwy programista-człowiek. No i przestrzeń możliwych programów była śmiesznie mała. Piszę o tym nieco naiwnym eksperymencie z pewnym rozczuleniem. Ja oczywiście, znając swój algorytm, mogłem przewidzieć jak się będzie zachowywał. A jednak zrobiło to na mnie silne wrażenie.

Linki
https://sourceforge.net/projects/dorban/
https://github.com/pawelbiernacki/svarog
https://sourceforge.net/projects/perkun/
https://sourceforge.net/projects/perkunwars/