Politechnika Warszawska

WYDZIAŁ MATEMATYKI
I NAUK INFORMACYJNYCH

Praca dyplomowa magisterska

na kierunku Informatyka

w specjalności Metody sztucznej inteligencji

Zestawienie technik uczenia maszynowego i heurystyk zastosowanych do proceduralnego generowania poziomów w grze Sokoban

Patryk Fijałkowski

Numer albumu 286350

promotor

dr inż. Maciej Bartoszuk

WARSZAWA 2021

....

podpis promotora   podpis autora

Streszczenie

Zestawienie technik uczenia maszynowego i heurystyk zastosowanych do proceduralnego generowania poziomów w grze Sokoban

Celem pracy jest wszechstronne porównanie czterech technik, które zostaną zastosowane do proceduralnego generowania poziomów w grze *Sokoban*. Zestawienie czterech fundamentalnie różnych technik pozwoli na wyznaczenie ich mocnych i słabych stron, dając tym samym podłoże do dalszego rozwoju badań nad generowaniem proceduralnym.

W pracy zostanie sprawdzone, jak konfigurowalne są poszczególne metody pod kątem trudności i rozmiaru poziomu. Zostanie zbadana efektywność czasowa każdej z metod. Przy użyciu autorskich metryk zostanie również określone, jak skomplikowane poziomy mogą zostać wygenerowane przez prezentowane techniki. *Sokoban* jest prostą grą łamigłówkową, w której gracz ma za zadanie przemieścić przedmioty w wyznaczone miejsca na planszy będącej prostokątną siatką, gdzie każde pojedyncze pole jest kwadratem. Trudność gry jest zależna od rozmieszczenia przedmiotów i przeszkód na planszy (poziomie). Zastosowanie proceduralnego generowania poziomów ma na celu odciążyć twórcę gry z wymyślania ich. Ponadto, generowanie proceduralne może potencjalnie tworzyć poziomy znacznie bardziej zawiłe niż wymyśliłby je człowiek.

Większość opisywanych w literaturze algorytmów wymaga czasochłonnego procesu uczenia agenta. Ten proces ma charakter inkrementalny i techniki oparte na nim należą do zbioru algorytmów uczenia maszynowego. Z tego powodu, poza dwoma technikami z tego działu zostaną również zaprezentowane dwie heurystyki, co pozwoli na zestawienie podejścia heurystycznego z uczeniem maszynowym. Ponadto, weryfikacji zostanie poddana wartość dodana płynąca z wiedzy eksperckiej w kontekście generowania poziomów.

Słowa kluczowe: Generowanie proceduralne, sztuczna inteligencja, heurystyka, *Sokoban*, uczenie maszynowe

Abstract

Comparison of machine learning techniques and heuristics used for procedural level generation in Sokoban game

TODO - dopiszę po zaakceptowaniu wersji polskiej

Keywords: Procedural content generation, artificial intelligence, heuristics, *Sokoban*, machine learning

Warszawa, dnia .....

Oświadczenie

Oświadczam, że pracę magisterską pod tytułem „Zestawienie technik uczenia maszynowego i heurystyk zastosowanych do proceduralnego generowania poziomów w grze Sokoban", której promotorem jest dr inż. Maciej Bartoszuk, wykonałem samodzielnie, co poświadczam własnoręcznym podpisem.

.....

Spis treści

1. Wstęp

Generowanie proceduralne to tworzenie zawartości przy użyciu algorytmów. Zamiast tworzyć zawartość ręcznie, można to zadanie zlecić wyspecjalizowanym metodom. Wykorzystanie komputerów do generowania proceduralnego może przynieść więcej treści o wyższej jakości niż ludzka kreacja manualna, dlatego w dobie dynamicznego rozwoju technologii jest to istotne zagadnienie. Wśród licznych przykładów wykorzystania generowania proceduralnego warto wymienić oprogramowanie *Massive* [11], które generowało tłumy liczące setki tysięcy postaci w bitwach z filmowej trylogii *Władcy Pierścieni* czy też *Terragen* [5], przy wykorzystaniu którego artyści i fotografowie dodają głębi przedstawianym krajobrazom.

Przemysł komputerowych gier wideo jest dynamicznie rozwijającą się gałęzią gospodarki. Wartość rynku gier, jako jednego z najmłodszych przemysłów kreatywnych, wyróżnia się monotonicznym tempem wzrostu [25]. W roku 2019 wyceniało się rynek gier na ponad 150 miliardów dolarów, a wzrost jego wartości w latach $2018 - 2022$ szacuje się na 27% [25]. Z tegorocznych analiz wynika, że branża gier przynosi więcej zysków niż filmowa i muzyczna razem wzięte [30]. Reasumując, aktualna sytuacja jednoznacznie wskazuje na to, że ten dział branży kreatywnej jest w rozkwicie.

1.1. Proceduralne generowanie zawartości w grach

Podczas tworzenia gier, zatrudniani są ludzie odpowiedzialni za tworzenie grafik, animacji czy projektowania poziomów. Zamiast tego, można posiłkować się odpowiednimi technikami do wygenerowania potrzebnego rodzaju zawartości.

Mówi się o zastosowaniu generowania proceduralnego w grach różnych gatunków. Przyjmuje się [20], że jest to technika powszechnie stosowana, która została zapoczątkowana w roku 1978, wraz z wydaniem *Beneath Apple Manor*. W tej produkcji zadaniem gracza jest sukcesywne przemierzanie kolejnych pokojów podziemnego świata, pokonując napotkanych na swojej drodze wrogów. Twórcy *Beneath Apple Manor* stworzyli algorytm generujący losowe pokoje, złożone z losowych kombinacji wrogów. Takie rozwiązanie powoduje, że każda rozgrywka będzie inna,

tym samym stawiając przed graczem różne wyzwania.

Koncept proceduralnego generowania zawartości jest odciążeniem twórców gier. Coraz powszechniejsze staje się wykorzystywanie technik uczenia maszynowego w tym celu, co wykazano w p. 1.1.1. Istnieją jednak pewne obostrzenia z tym związane, na przykład czasowe. Techniki uczenia maszynowego najczęściej mają charakter iteracyjny i wymagają dużej liczby iteracji w celu osiągnięcia zadowalających wyników. Wobec tego, nieakceptowalne jest użycie tego rodzaju algorytmów w czasie rzeczywistym w grach. Zamiast tego, dokonuje się selekcji i systematyzacji wygenerowanej zawartości i dostarcza się ją wraz z grą [13].

1.1.1. Przykłady

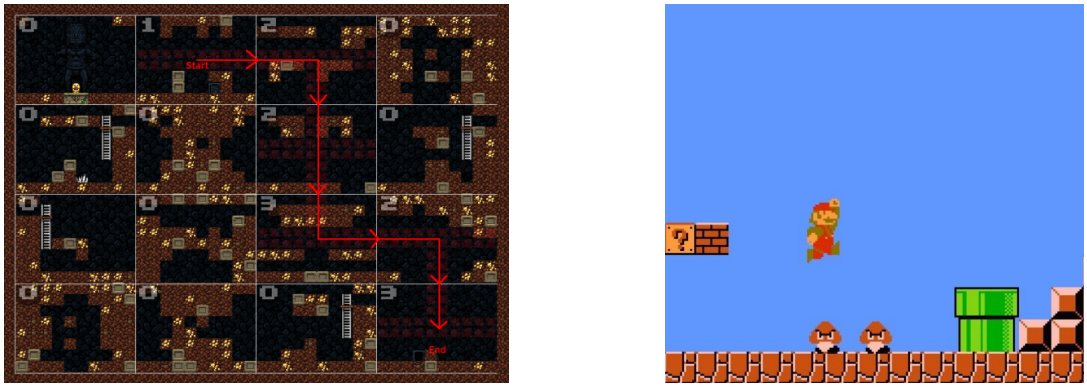
Interesującym zastosowaniem uczenia maszynowego w generowaniu proceduralnym jest gra *Galactic Arms Race* [8] wydana w 2014 roku przez *Evolutionary Games*. Twórcy posiłkowali się autorskim algorytmem genetycznym, tworząc różne konfiguracje pocisków statków kosmicznych, takich jak na rys. 1.1. Algorytm *cgNEAT* [8] ewoluuje, mając na uwadze preferencje użytkownika, które są określane na bieżąco podczas rozgrywki. Oznacza to, że twórcy nie musieli kreować działania i graficznych aspektów pocisków – ta zawartość była tworzona przez algorytm.

Rysunek 1.1: Zrzut ekranu z gry *Galactic Arms Race*, źródło: http://indiedb.com

W grze *Spelunky* zadaniem gracza jest przemierzanie kolejnych poziomów, co wymaga zręczności i szybkiego podejmowania decyzji. Twórcy *Spelunky* zdecydowali się na rozwiązanie hybrydowe w kontekście generowania poziomów [28]. Algorytm jest heurystyką, która działa w oparciu o podział poziomu na 16 części, jak na rys. 1.2a. Każda z nich jest tworzona na podstawie sza-

blonu (ang. *template*), których zbiór został stworzy przez projektantów. Algorytm poprawia tak stworzony poziom, dodając element losowości i zapewniając, że istnieje ścieżka od wejścia do wyjścia z poziomu.

Generowanie proceduralne nie jest techniką stosowaną wyłącznie w nowych grach. Autorzy [27] wykorzystali generatywne sieci współzawodniczące (ang. *Generative Adversarial Networks*) do tworzenia nowych poziomów w kultowej grze *Super Mario Bros.*, pochodzącej z 1983 roku.



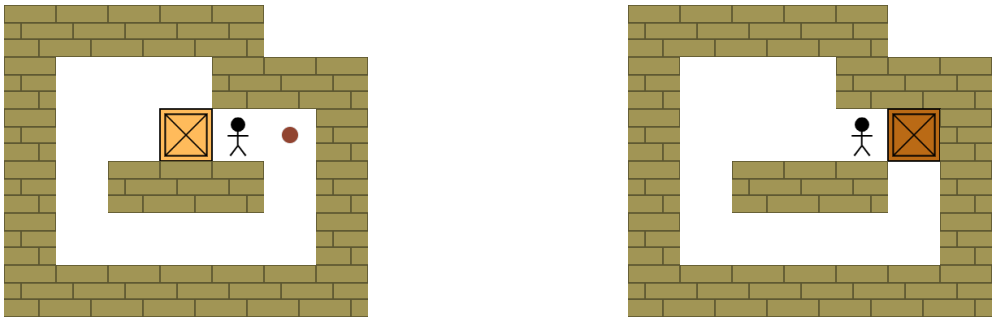
(a) Poziom w *Spelunky* (b) Poziom w *Super Mario Bros.*

Rysunek 1.2: Poziomy w grach korzystających z generowania proceduralnego, źródła: [28], [27]

Powszechnym podejściem w generowaniu proceduralnym są algorytmy generowania szumu gradientowego. Szczególnie często wykorzystuje się takie rozwiązania w grach z gatunku *sanbox*, które przedstawiają graczom otwarty świat, nie narzucając sposobów jego eksploracji i pozostawiając dowolność w kwestii interakcji z jego elementami. Gry takie jak *Terraria* czy *Minecraft*, tworzą dla graczy potencjalnie nieskończone światy dwu- lub trójwymiarowe.

1.2. Sokoban

Sokoban jest grą logiczną, rozgrywaną na planszy złożonej z kwadratów. Celem gracza jest przesunięcie wszystkich pudeł w wyznaczone kwadraty docelowe, tak jak na rysunku 1.3b. Zaczynając z planszą jak na rysunku 1.3a, gracz musi wykonać co najmniej 8 ruchów, w tym 4 pchnięcia pudeł. Jednak istnieje też rozwiązanie tej planszy, wykorzystujące 12 ruchów, w tym 2 pchnięcia. Różne implementacje gry uznają liczbę ruchów albo liczbę pchnięć jako ocenę rozwiązania.



(a) Plansza początkowa

(b) Plansza końcowa

Rysunek 1.3: Przykładowa plansza *Sokoban*

Oryginalnie *Sokoban* jest grą wideo autorstwa Hiroyuki Imabayashi, wydaną w 1982 roku na platformę PC-88. Od tamtego czasu koncepcja gry jest rozwijana i implementowana w innych grach, takich jak *Pokémon Emerald* czy *Grand Theft Auto: San Andreas*. Ponadto, użytkownicy tworzą coraz bardziej skomplikowane plansze, czego dowodem są zbiory wymienione w tabeli 1.1.

Tabela 1.1: Zbiory plansz Sokoban, źródło: [31]

Nazwa zbioru	**Liczba poziomów**
Sven	1911
Sasquatch I-VII	350
Sokoban Perfect	306
Sokoban Revenge	306
Aymeric	282
SokHard	163

1.2.1. Rozwiązanie

Problem podania rozwiązania czy choćby ustalenia, czy zadana plansza *Sokoban* jest rozwiązywalna, jest problemem *PSPACE*-zupełnym [14]. Problemy *PSPACE*-zupełne to grupa problemów, które mogą zostać rozwiązane przy użyciu maszyny Turinga, wykorzystując wielomianową ilość pamięci, ale nie są problemami NP.

Drzewo decyzyjne odpowiadające rozgrywce, mimo swojego niskiego współczynnika rozgałęzienia (ang. *braching factor*), jest wysokie. W przypadku rozwiązywania planszy *Sokoban*, omawiany współczynnik wynosi co najwyżej cztery, ponieważ w każdej sytuacji gracz może dokonać ruchu w jednym z czterech kierunków.

W związku z tym, powstają heurystyczne *solvery*, które potrafią rozwiązywać plansze *Soko-*

ban w czasie wielomianowym [31]. Mimo to, nie powstał jeszcze *solver*, który rozwiązywałby poprawnie wszystkie plansze z badanego zbioru [31].

Problem rozwiązania planszy *Sokoban* jest powiązany z wieloma metodami generowania plansz. Metody generujące, które nie opierają swojego działania na zasymulowaniu rozgrywki, potrzebują na różnych etapach swojego działania weryfikować, czy aktualnie tworzona plansza jest rozwiązywalna. Jako że sprawdzenie tego w sposób dokładny nie jest możliwe czasowo, w tym celu również wykorzystywane są heurystyczne *solvery*.

2. Prezentacja metod

W niniejszej pracy zdecydowano się zestawić cztery metody generowania poziomów *Sokoban*, pogrupowanych w tabeli 2.1. Dogłębna analiza metod proponowanych w literaturze dla problemu generowania poziomów *Sokoban* pozwoliła na dobór reprezentatywnych metod. Wybrano dwie będące heurystykami, które nie korzystają z technik uczenia maszynowego, oraz dwie – korzystające. Ponadto, dobrano dwie, które bazują na symulacji rozgrywki oraz dwie, które działają bez symulowania działań gracza.

Istotną zaletą analizowanych metod, które symulują rozgrywkę, jest możliwość podania przykładowej sekwencji ruchów, rozwiązujących daną planszę – tej z symulacji. Z kolei metody, w których nie symuluje się rozgrywki, mogą zostać zastosowane do innych problemów niż generowanie plansz *Sokoban*, ze względu na swoje uogólnienie analizowanego problemu. Autorzy [1] wprowadzają swoją metodę, stosując ją jeszcze do generowania plansz dwóch innych rodzajów. Autorzy [6] podkreślają, że ich metodę generacji można zaadaptować do każdego problemu, który da się sprowadzić do postaci opisanej w p. 2.2.1.

Tabela 2.1: Analizowane metody

	Heurystyka	**Uczenie maszynowe**
Symulacja rozgrywki	SYM [29]	MCTS [4]
Brak symulacji	PDB [6]	PPO [1]

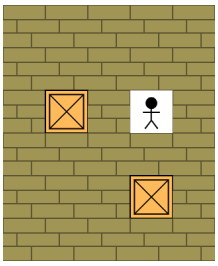
2.1. Metoda SYM

Metoda SYM [29], opiera swoje działanie na symulowaniu zmodyfikowanej rozgrywki *Sokoban*. Iteracyjnie wykonując pewne akcje i przechodząc po początkowo pustej planszy, agent wyznacza wolne pola na planszy.

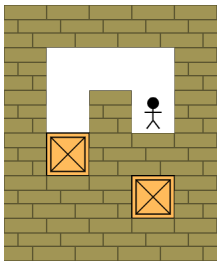
2.1.1. Opis metody

Metoda SYM rozpoczyna pracę, losując początkowe ustawienia pudeł na planszy. Następnie agent wykonuje założoną liczbę razy dwa rodzaje rozgrywek: najpierw rozgrywki typu *forward*, a następnie – *backward*, by potem uporządkować generowaną planszę.

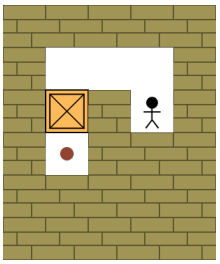
W rozgrywce *forward*, agent wybiera w określony sposób pudło i kierunek jego przepychania, by dokonać przepchnięcia. Jeśli przepchnięcie jest możliwe, to pudło zmienia swoją pozycję, a pola przez które przeszedł agent, stają się pustymi polami. Przykładowo, jeśli dla planszy na rys. 2.1a gracz wybierze wyższe pudło i dół jako kierunek przepychania, to jego akcja spowoduje że plansza przybierze układ jak na rys. 2.1b.



(a) Plansza przed wykonaniem akcji



(b) Plansza po wykonaniu akcji

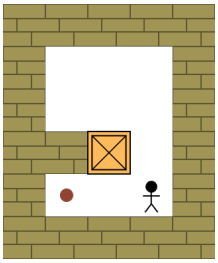


(c) Plansza po zakończeniu rozgrywki

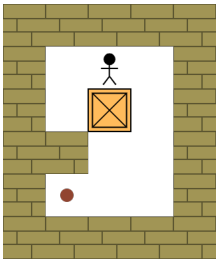
Rysunek 2.1: Działanie akcji w rozgrywce *forward*

Po zakończeniu rozgrywki *forward*, pudła zamieniane są w pola docelowe, a startowe pozycje pudeł stają się faktycznymi pudłami. Ponadto, nieprzepchnięte pudła zamieniane są w ściany. Kontynuując analizę przypadku, jeśli rozgrywka *forward* zostanie zakończona z planszą jak na rys. 2.1b, to w wyniku transformacji otrzyma się planszę przedstawioną na rys. 2.1c i agent przejdzie do przeprowadzenia rozgrywki *backward*, w celu dodatkowego skomplikowania planszy.

W rozgrywce *backward*, agent wybiera w określony sposób pudło i kierunek jego przeciągania, by dokonać przeciągnięcia. Jeśli przeciągnięcie jest możliwe, to pudło oraz gracz aktualizują swoje pozycje. Przykładowo, jeśli dla planszy na rys. 2.2a gracz wybierze jedyne dostępne pole docelowe i lewo jako kierunek przepychania, to jego akcja spowoduje że plansza przybierze układ jak na rys. 2.2b.



(a) Plansza przed wykonaniem akcji



(b) Plansza po wykonaniu akcji

Rysunek 2.2: Działanie akcji w rozgrywce *backward*

2.1.2. Strategie

Wybór pudła	**Kierunek przepychania**	**Kierunek przeciągania**	**Wybór ścieżki**
Random	*Random*	*Random*	*Direct*
Least Pushed	*Farthest*	*Farthest*	*Closest Active Tile*
In Order	*Most Obstacles*	*Most Obstacles*	
		Most Access	

Tabela 2.2: Strategie decyzyjne metody SYM

W procesie generowania planszy metody SYM, agent podejmuje decyzje o przepychaniu i przeciąganiu pudeł oraz o sposobie wyboru ścieżek. Wybory dokonywane są zgodnie z ustalonymi wcześniej strategiami, będącymi hiperparametrami metody, które zostały zaprezentowane w tab. 2.2. Strategie *Random* dokonują wyborów w sposób losowy. Strategia *Least Pushed* wybiera te pudła, które w momencie podejmowania decyzji zostały przepchnięte najmniejszą liczbę razy, a *In Order* – wybiera pudła po kolei, niezależnie od liczby przepchnięć. *Farthest* maksymalizuje odległość od pudła do jego pola docelowego. *Most Obstacles* wybiera kierunek tak, by wynikowa pozycja pudła sąsiadowała z jak największą liczbą ścian i innych pudeł. *Most Access* ma zapewnić agentowi jak najwięcej dostępnych kierunków po wykonaniu akcji przeciągania. Strategia wyboru ścieżek *Closest Active Tile* stara się wykorzystać jak najwięcej już

odwiedzonych pól, zachowując w ten sposób więcej ścian niż jej odpowiadająca strategia *Direct*, która wybiera najkrótsze możliwe ścieżki.

2.2. Metoda PDB

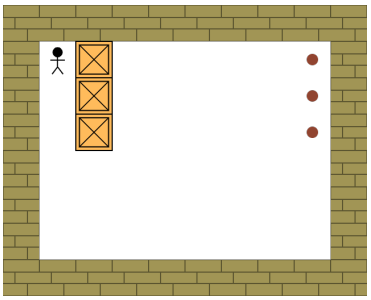
Metoda PDB [6] jest wprowadzona w oparciu o teorię problemu przeszukiwań przestrzeni stanu (ang. *state-space search problem*) oraz baz danych wzorców (ang. *pattern databases*).

2.2.1. Problem przeszukiwań przestrzeni

Problemem przeszukiwań przestrzeni stanu nazywa się krotkę opisaną w równaniu 2.1. Składowe problemu to kolejno zbiór stanów S, stan początkowy s_0, zbiór stanów docelowych S^* oraz zbiór akcji A. Stan definiuje się jako pełne przypisanie wartości wszystkim zmiennym $v \in V$, przy zachowaniu dziedzin D_v. Rozwiązaniem problemu przeszukiwań przestrzeni jest ścieżka rozwiązująca (ang. *solution path*) – skończony ciąg akcji, po wykonaniu którego ze stanu s_0 otrzyma się stan ze zbioru S^*.

$$\mathcal{P} =< S, s_0, S^*, A > \tag{2.1}$$

Opisywany problem różni się od innych problemów przeszukiwania tym, że przestrzeń stanów jest niejawna (ang. *implicit*). W prezentowanym rozwiązaniu, graf przestrzeni stanów jest zbyt duży, aby można go było wygenerować i przechowywać w pamięci. Zamiast tego, jedynie interesujące węzły są generowane i robione jest to na bieżąco – w trakcie ich eksploracji. Stanami w metodzie PDB są przypisania lokalizacji k pudeł i gracza, a więc składają się z $k+1$ zmiennych. Dziedziną każdej zmiennej są puste pola. W związku z tym, liczba możliwych stanów opisana jest wzorem $(n-k)\binom{n-k-1}{k}$, gdzie n to liczba wolnych pól. Przykładowo, dla planszy małych rozmiarów przedstawionej na rys. 2.3 istnieje aż 595 980 możliwych stanów (wierzchołków grafu).



Rysunek 2.3: Przykładowa plansza o rozmiarze 8 x 10

Autorzy [6] definiują ponadto problem wygenerowania początkowego stanu (ang. *initial state generation task*) $\mathcal{P}_{-s_0}$, opisany krotką w równaniu 2.2. Jest on równoważny problemowi przeszukiwań przestrzeni ze stanem początkowym s_0, jednak celem nie jest wyznaczenie ścieżki,

a wskazanie stanu $s \in S$, dla którego problem $\mathcal{P}_{-s_0}$ z s jako stanem początkowym, jest rozwiązywalny.

$$\mathcal{P}_{-s_0} =< S, S^*, A > \tag{2.2}$$

2.2.2. Bazy danych wzorców

W celu rozwiązania problemu opisanego w p. 2.2.1, stosuje się heurystykę bazy danych wzorców, wprowadzoną przez Josepha Culbersona oraz Jonathana Schaeffera [15]. Idea tej heurystyki oparta jest na zignorowaniu części problemu (podzbioru zmiennych V) i wyznaczeniu dolnego ograniczenia długości ścieżki optymalnej dla pozostałych zmiennych. Jako lepsze ograniczenie stosuje się sumę wartości heurystyk dla rozłącznych podzbiorów zbioru zmiennych [3].

2.2.3. Opis metody

Metoda PDB korzysta z algorytmu β, który rozwiązuje problem opisany wzorem 2.2 w oparciu o podejście zachłanne. W tym celu wykorzystywane jest przeszukiwanie grafu, które priorytetyzuje węzły (ang. *best-first search*) zgodnie z zadaną sekwencją $[\cdot]$. Sekwencja $[\cdot]$ złożona jest z wyselekcjonowanych wartości heurystyk wartościujących dany stan. Rodzaje wykorzystywanych heurystyk zostały wymienione poniżej. Autorzy metody wykorzystują różne konfiguracje sekwencji $[\cdot]$, co zostało opisane dokładniej w podrozdz. 3.3.

- $w(h)$ - funkcja nowości,
- h^{PDB} - heurystyka bazy danych wzorców,
- kC - liczba konfliktów rzędu k.

Podczas eksploracji, algorytm korzysta z funkcji nowości $w(h)$ (ang. *novelty function*) [21], która zwraca najwyższe wartości dla stanów, w których zmienne przyjmują nieużywane wcześniej wartości. Heurystyki h^{PDB} są tożsame z długością pewnej ścieżki rozwiązującej. Warto zaznaczyć, że współczynnik korelacji między długością optymalnej ścieżki a czasem potrzebnym do rozwiązania planszy *Sokoban* w teście rang Spearmana wynosi 0.47 [22]. Trzecim rodzajem używanych heurystyk jest liczba konfliktów rzędu k, która jest równa różnicy wartości heurystyki bazy danych dla wzorca rozmiaru k i $k-1$, przyjmując jedynie wartości nieujemne. Intuicyjnie – kC nagradza stany, w których istnieją zależności między k zmiennymi.

Graf stanów jest przeszukiwany wstecz, od wierzchołków odpowiadającym stanom docelowym, co gwarantuje istnienie ścieżki docelowej. Autorzy metody PDB podkreślają, że prze-

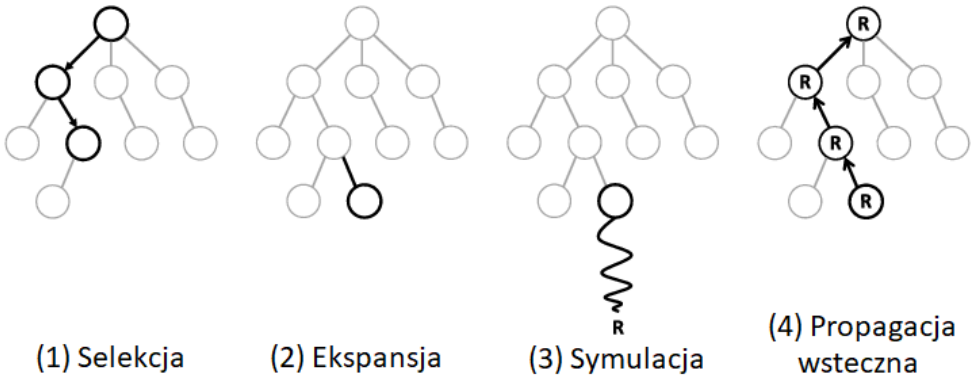
szukiwanie grafu od wierzchołka początkowego s_p wymagałoby zapewnienia o istnieniu ścieżki docelowej rozpoczynającej się w s_p, co jest wymagające obliczeniowo [14].

2.3. Metoda MCTS

Autorzy wprowadzają metodę MCTS [4] jako autorską adaptację heurystyki *Monte-Carlo Tree Search*.

2.3.1. Wprowadzenie

Monte-Carlo Tree Search jest heurystyką, której celem jest podejmowanie decyzji w pewnych zadaniach sztucznej inteligencji. Metoda opiera swoje działanie na przeszukiwaniu możliwych stanów zapisanych w wierzchołkach drzewa i losowym symulowaniu rozgrywek. Algorytmy z grupy MCTS opierają się na iteracyjnym rozbudowywaniu drzewa stanów przez sekwencyjne wykonanie czterech faz – selekcji, ekspansji, symulacji i propagacji wstecznej. Poszczególne fazy zostały zobrazowane na rys. 2.4.



Rysunek 2.4: Fazy MCTS, źródło: [26]

Faza selekcji w heurystyce *Monte-Carlo Tree Search* pozostawia dowolność w wyborze liścia, który będzie eksploatowany w kolejnych fazach. Oczywiście sposób wybierania liści w kolejnych iteracjach jest krytyczny z punktu widzenia eksploracji drzewa i działania metody. W literaturze opisywane są różne podejścia, przykładowo UCB_Minimal w [10], czy UCB-V w [12]. Najbardziej powszechny i użyty w metodzie MCTS jest jednak wariant UCT, opisany w [18]. Ten wariant stara się zachować równowagę między eksploatacją bardziej obiecujących ruchów a eksploracją tych rzadko odwiedzonych. Formuła, która odpowiada za wyznaczenie najbardziej obiecującego wierzchołka w fazie wyboru MCTS jest przedstawiona jako wyrażenie (2.3). Indeks i odnosi się do liczby wykonanych przez algorytm iteracji, czyli czterech faz MCTS. W pierwszym składniku sumy wyrażenia (2.3), licznik w_i oznacza sumę wszystkich wypłat w danym węźle, a mianownik n_i oznacza liczbę rozegranych symulacji. Parametr eksploracji c, może być dostosowany do badanego problemu. Odwołując się do [18], dla problemu z wypłatami w przedziale $[-1, 1]$, optymalnie jest przyjąć $c = \sqrt{2}$.

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln\left(\sum_i n_i\right)}{n_i}} \tag{2.3}$$

2.3.2. Opis metody

Metoda MCTS opiera swoje działanie na symulowaniu zmodyfikowanej rozgrywki Sokoban, która składa się z dwóch faz. W pierwszej gracz ma do wyboru trzy akcje: może usuwać przeszkody graniczące z pustymi polami, stawiać pudła lub zamrozić planszę. Algorytm zaczyna pracę z planszą w pełni wypełnioną przeszkodami, więc zadaniem fazy pierwszej jest przygotowanie pustych pól. W wyniku akcji zamrożenia, rozgrywka przechodzi do fazy drugiej, w której gracz prowadzi normalną rozgrywkę, poruszając się i przesuwając pudła, do momentu aż zdecyduje się ją zewaluować. Ewaluacja jest akcją, która kończy rozgrywkę i dokonuje dodatkowych procesów czyszczenia w celu uzyskania maksymalnie dobrej i poprawnej planszy.

Kluczowe dla metody MCTS jest dobranie funkcji oceniającej jakość generowanych poziomów. Jako że *Monte-Carlo Tree Search* opiera swoje działanie na wielokrotnym symulowaniu rozgrywek, wyznaczanie wypłaty dla danego poziomu musi być mało kosztowne obliczeniowo. Autorzy [4] zdecydowali się na funkcję opisaną wzorem 2.4, która jest sumą ważoną trzech metryk, podzieloną przez stałą z, dla znormalizowania wyniku. Wyrazy w_b, w_c i w_n są wagami dla odpowiednich metryk, opisanych poniżej.

1. Metryka pudeł 3 x 3 P_b - liczba pól, która nie znajduje się w bloku 3 x 3 przeszkód lub pustych pól. Zadaniem tej metryki jest nagrodzenie plansz, które nie zawierają bloków 3 x 3 tych samych pól, gdyż tego typu czynią plansze mniej skomplikowanymi.

2. Metryka zagęszczenia P_c - funkcja nagradzająca plansze o długich ścieżkach między pudłami a ich docelowymi punktami.

3. Metryka pudeł P_n - liczba pudeł na planszy.

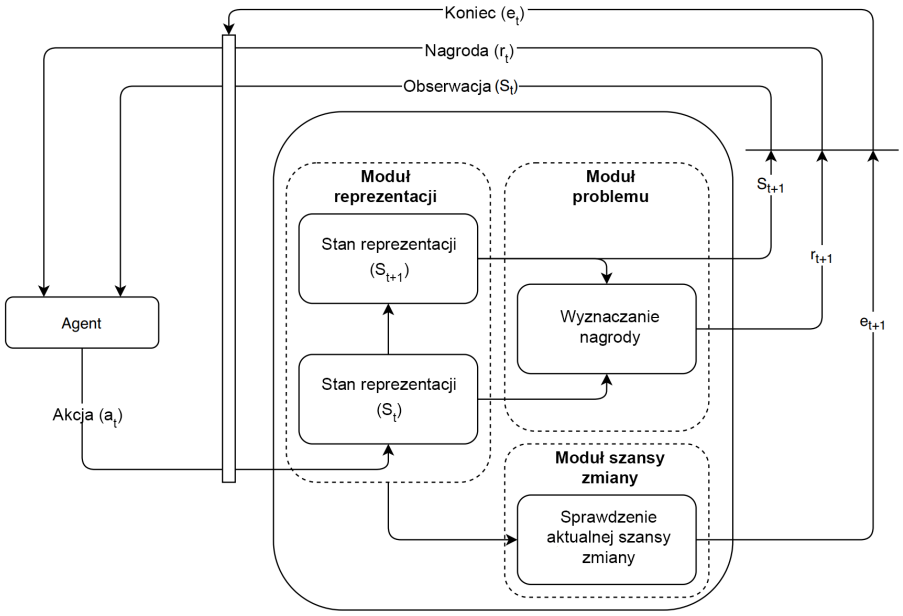
$$\frac{w_b P_b + w_c P_c + w_n P_n}{z} \tag{2.4}$$

2.4. Metoda PPO

2.4.1. Opis metody

Metoda PPO [1] opiera swoje działanie na algorytmie *Proximal Policy Optimization* uczeniu ze wzmocnieniem. Autorzy dzielą schemat jej działania na trzy moduły: problemu (ang. *problem module*), reprezentacji (ang. *representation module*) i szansy zmiany (ang. *change percentage*), co zostało ukazane na schemacie z rys. 2.5. Dokładniejszy opis poszczególnych modułów został przedstawiony w rozdziale 2.4.2.

Agent na podstawie obserwacji aktualnego stanu S_t dokonuje akcji a_t, w wyniku czego moduł reprezentacji przekształca S_t w S_{t+1}. Wtedy moduł problemu wyznacza nagrodę r_{t+1}, zestawiając ze sobą oba stany. Potem do agenta przekazywane są S_{t+1} oraz r_{t+1} i pod warunkiem, że agent nie otrzymał informacji e_{t+1} o przekroczeniu dozwolonej liczby zmian, wykonywane są dalsze iteracje uczenia.



Rysunek 2.5: Schemat działania metody PPO, źródło: [1]

2.4.2. Moduły

Moduł problemu dostarcza informacji na temat rozmiaru planszy i dostępnych typów obiektów. Ten moduł ocenia też, jak akcje podejmowane przez agenta wpływają na jakość generowanych poziomów. Ponadto, zadaniem tego modułu jest też stwierdzenie, kiedy epizod uczenia powinien zostać zakończony.

Moduł reprezentacji inicjalizuje problem, przechowuje aktualny stan i modyfikuje go, opie-

rając się na akcjach agenta. W prezentowanym zastosowaniu, każdej komórce na planszy odpowiada wartość z zakresu $[0, 6]$. Zaimplementowane są trzy różne reprezentacje, opisane poniżej. Najistotniejszą różnicą jest zasięg zmian w każdej reprezentacji – wąska ma z góry narzuconą sekwencję pól, żółwia na każdym kroku ma maksymalnie 4 dostępne, a szeroka ma pełną dowolność. Mówi się że reprezentacja szeroka ma globalny zasięg zmian w przeciwieństwie do reprezentacji wąskiej i żółwiej.

1. Wąska – na każdym kroku agent obserwuje aktualną planszę i pewną lokację na planszy. Może wtedy zamienić wartość komórki w zadanej lokacji, ale może też przejść do dalszych kroków bez tej zmiany.

2. Żółwia – na każdym kroku agent obserwuje aktualną planszę i lokację. Może zamienić wartość komórki w aktualnej lokacji albo przemieścić aktualną lokację w dowolne sąsiadujące z nią pole. Ten sposób reprezentacji jest inspirowany językami programowania Logo, w których użytkownik przemieszcza wirtualnego żółwia po planszy.

3. Szeroka - na każdym kroku agent obserwuje aktualną planszę i może zamieniać wartości komórek w dowolnych lokacji.

Moduł szansy zmiany ogranicza możliwość zmieniania dowolnych komórek na planszy, żeby agent nie wymienił wszystkich zadanych komórek, limitując tym samym długość jednego epizodu. Zgodnie z obserwacjami opisanymi w [1], agent o niskiej szansie zmiany wykonuje akcje zachłanne, maksymalizując wypłaty w krótkiej perspektywie, podczas gdy agent o wysokiej szansie zmiany zdaje się wyznaczać bardziej optymalne i długoterminowe plansze. Problemem z agentem o wysokiej szansie zmiany jest ryzyko dążenia do jednej, najbardziej korzystnej planszy, co nie jest celem tej metody. W tym przypadku agenta chce się nauczyć poprawiania zadanych plansz na lepsze.

2.4.3. Weryfikacja planszy

Opisany w rozdziale 2.4.2 moduł problemu weryfikuje poprawność generowanych plansz, przy użyciu naiwnego *solvera*, bazującego na przeszukiwaniu drzewa rozgrywki aż do 5000 wierzchołków. W celu wyszukiwania najkrótszej ścieżki, użyty został algorytm A*.

2.4.4. Implementacja

Implementacja, na bazie której przeprowadzono eksperymenty, została przygotowana w języku *Python* w wersji 3.5. Wykorzystano biblioteki *Stable Baselines* w wersji 2.9 oraz *OpenAI Gym* w wersji 0.7.4.

Agentowi aktualna plansza przedstawiana jest przy użyciu one-hot-encoding. Użyto dwóch różnych sieci neuronowych. Pierwsza z nich jest wykorzystana do reprezentacji żółwiej i wąskiej. Druga do reprezentacji szerokiej z powodu na istotnie większą przestrzeń dostępnych akcji.

1. Architektura 1 - złożona z trzech warstw konwolucyjnych i dwóch w pełni połączonych. Ta architektura była oparta na [19].

2. Architektura 2 - złożona z dziesięciu warstw konwolucyjnych i dwóch w pełni połączonych. Ta architektura była oparta na [7].

3. Wyniki eksperymentów

3.1. Maszyna testowa

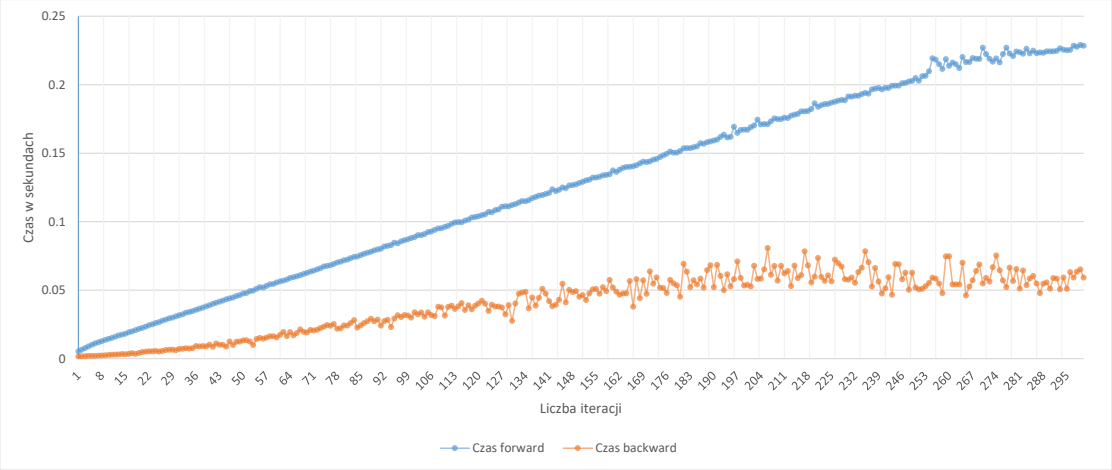
Wszystkie eksperymenty zostały wykonane na komputerze:

- z zainstalowanym systemem operacyjnym *Windows 10 Education N*,
- wyposażonym w procesor *Intel Core i7-8700k @3.70 GHz*,
- wyposażonym w kartę graficzną *NVIDIA GeForce GTX 1060 6GB*,
- wyposażonym w 32GB pamięci RAM.

3.2. Metoda SYM

3.2.1. Wydajność czasowa

W celu zbadania wydajności czasowej metody SYM, wykonano eksperyment badający średni czas wykonania zadanej liczby iteracji *forward* oraz *backward*. W eksperymencie uwzględniono wszystkie możliwe konfiguracje strategii oraz rozmiary plansz z zakresu $[5, 30]$. Wyniki analizy, ukazane na rys. 3.1, wykazały iż średnio rozgrywka *forward* nie trwała dłużej niż 0.25 sekundy, a rozgrywka *backward* – 0.1 sekundy. Co więcej, dla liczby iteracji poniżej stu, generacja planszy zajmowała średnio mniej niż 0.122 sekundy.

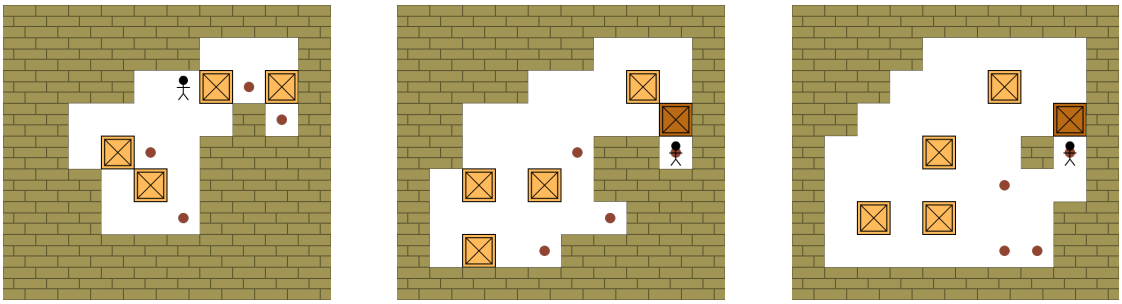


Rysunek 3.1: Zależność czasu od liczby iteracji

Autorzy metody SYM zestawiają wydajność czasową swojej metody z algorytmem zaprezentowanym w [16], który również jest metodą heurystyczną symulującą rozgrywkę. Metoda SYM wygrywa ze swoim odpowiednikiem przede wszystkim ze względu na niezależność od liczby pudeł na generowanej planszy. Metoda opisana w [16] dla liczby pudeł poniżej sześciu uzyskuje podobne wyniki, za to zależność czasu od liczby pudeł powyżej sześciu ma charakter wykładniczy. Co więcej, czas obliczeń odpowiednika metody SYM rośnie kwadratowo w zależności od wartości hiperparametru definiującego skomplikowanie plansz, czego nie obserwuje się w przypadku metody SYM. Analizując inne badania zaprezentowane w [29], można wnioskować iż jest to najbardziej wydajna czasowo opublikowana metoda heurystyczna generowania plansz *Sokoban*.

3.2.2. Liczba iteracji

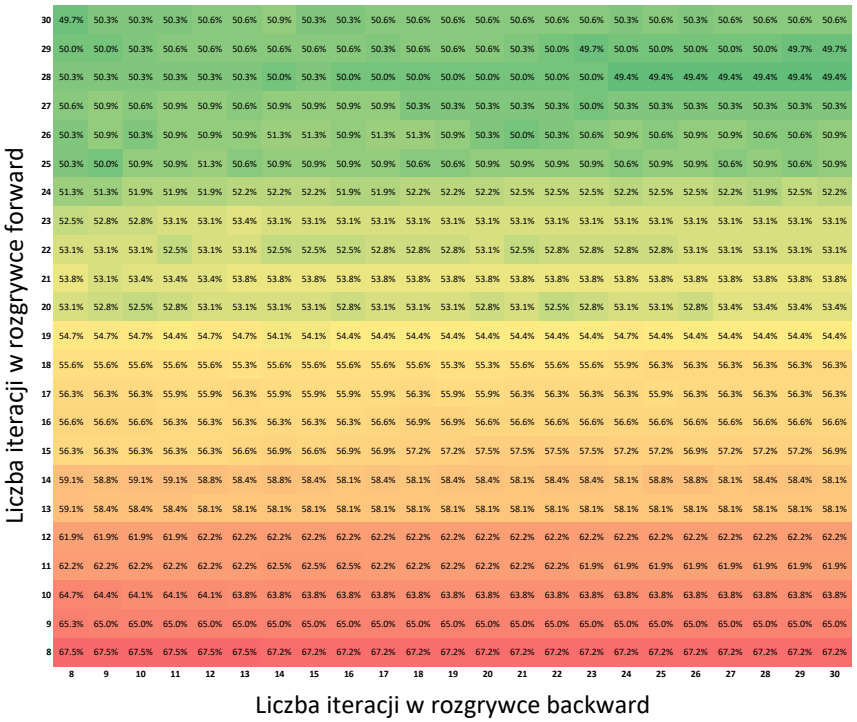
Wraz ze zwiększaniem liczby iteracji *forward* w metodzie SYM, plansze zyskują coraz więcej pustych pól, co można zaobserwować na rys. 3.2. To oznacza, że zmniejsza się ich wypełnienie – stosunek niepustych pól do wszystkich pól na planszy. Autorzy [22] sugerują, że większość



(a) Plansza po 4 iteracjach (b) Plansza po 10 iteracjach (c) Plansza po 30 iteracjach

Rysunek 3.2: Wpływ liczby iteracji *forward* na wypełnienie generowanych plansz

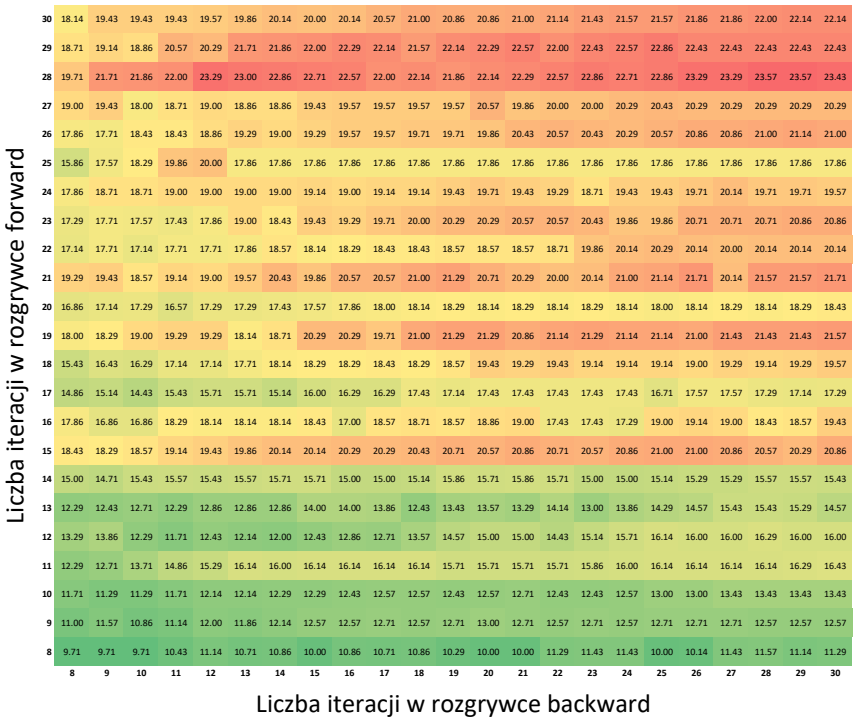
istotnie skomplikowanych plansz charakteryzuje się wypełnieniem na poziomie od 50% do 55%. Aby sprawdzić charakter wypełnienia dla analizowanej metody, dokonano generacji 5000 plansz rozmiaru $8x8$ z użyciem różnych kombinacji parametrów, dla określonego zakresu liczby iteracji *forward* i *backward*. Wyniki analizy zaprezentowano na rys. 3.3. Jak widać, dla tego rozmiaru planszy, żądane wypełnienie uzyskuje się dla liczby iteracji *forward* z zakresu $[17, 30]$. Liczba iteracji *backward* nie ma istotnego wpływu na wypełnienie generowanych plansz.



Rysunek 3.3: Zależność wypełnienia generowanych plansz od liczby iteracji

Jako że samo wypełnienie nie jest gwarantem skomplikowania poziomów, postanowiono zbadać również zależność średniej sumy odległości pudeł od ich pól docelowych. Wykonano eksperyment analogiczny do poprzedniego, a jego wyniki zaprezentowano na rys. 3.4. Jak widać, niezależnie od użytej liczby iteracji *forward*, wraz ze wzrostem liczby iteracji *backward*, sumy

odległości rosły, czyniąc plansze bardziej skomplikowanymi. Dla niektórych przypadków, zwiększenie liczby iteracji *backward*, wydłużało sumę odległości o ponad 40%.

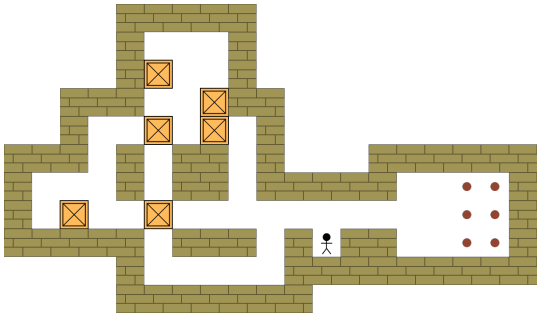


Rysunek 3.4: Zależność sumy odległości między pudłami a polami docelowymi od liczby iteracji

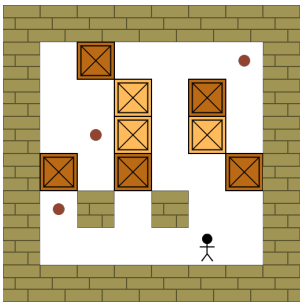
3.3. Metoda PDB

Kluczowe z punktu widzenia metody PDB jest dobranie odpowiedniej sekwencji $[\cdot]$. Autorzy metody, analizując jej wyniki, wyznaczyli 6 wariantów opartych na 20 różnych sekwencjach $[\cdot]$. W niniejszej pracy postanowiono przebadać 3 warianty o najlepszych wynikach, z których każdy oparty będzie na 4 sekwencjach. Takie podejście pozwala na bardziej kompleksowe zbadanie skuteczności metody w różnych konfiguracjach. Poniższe opisy badanych wariantów uwzględniają wielkości wzorców k z przedziału $[2, 5]$.

1. Wariant prosty – nie korzysta z funkcji nowości. Przyjęte sekwencje są postaci $\left[kC, h^{PDB_k}\right]$.
2. Wariant nowości – korzysta z funkcji nowości. Przyjęte sekwencje są postaci $\left[w(h^{PDB_k}), h^{PDB_k}\right]$.
3. Wariant złożony – korzysta z funkcji nowości oraz heurystyki konfliktów. Przyjęte sekwencje są postaci $\left[w(h^{PDB_k}), kC, h^{PDB_k}\right]$.



(a) Plansza ze zbioru*XSokoban*



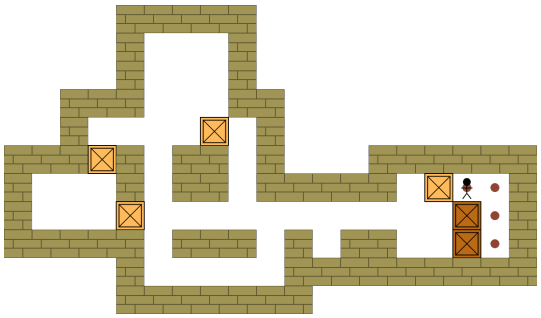
(b) Plansza ze zbioru *SokEvo*

Rysunek 3.5: Przykładowe plansze z użytych zbiorów

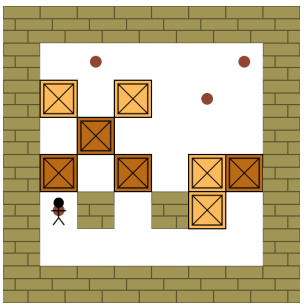
Badania autorów metody PDB zostały również poszerzone o dodatkowy zbiór plansz. Poza zbiorem *XSokoban* złożonym z 90 plansz, postanowiono wykorzystać zbiór *SokEvo*, na który składa się 107 poziomów [31]. Zasadnicza różnica między tymi zbiorami polega na tym, że żaden z poziomów w pierwszym z nich nie jest kształtu prostokąta, zaś w drugim – wszystkie. Przykładowe plansze z obu zbiorów wraz odpowiadającymi im wynikami metody ukazano na rys. 3.5 i 3.6.

3.3.1. Wydajność pamięciowa

Algorytm β opisany dokładnie w p. 2.2.3 korzysta z kolejki priorytetowej, w której przetrzymuje analizowane stany (plansze). Zaobserwowano, że wraz z czasem działania metody jej



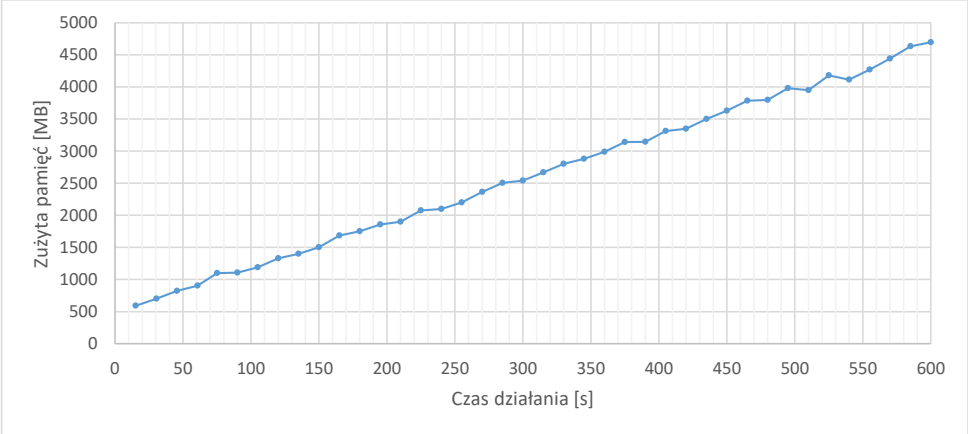
(a) Plansza ze zbioru *XSokoban*



(b) Plansza ze zbioru *SokEvo*

Rysunek 3.6: Przykładowe wyniki metody PDB oparte na planszach z rys. 3.5

zapotrzebowania pamięciowe rosną liniowo. Wyniki analiz zaprezentowano na rys. 3.7. Jak widać, 10 minut działania metody jest związane ze zużyciem prawie 5 GB pamięci.



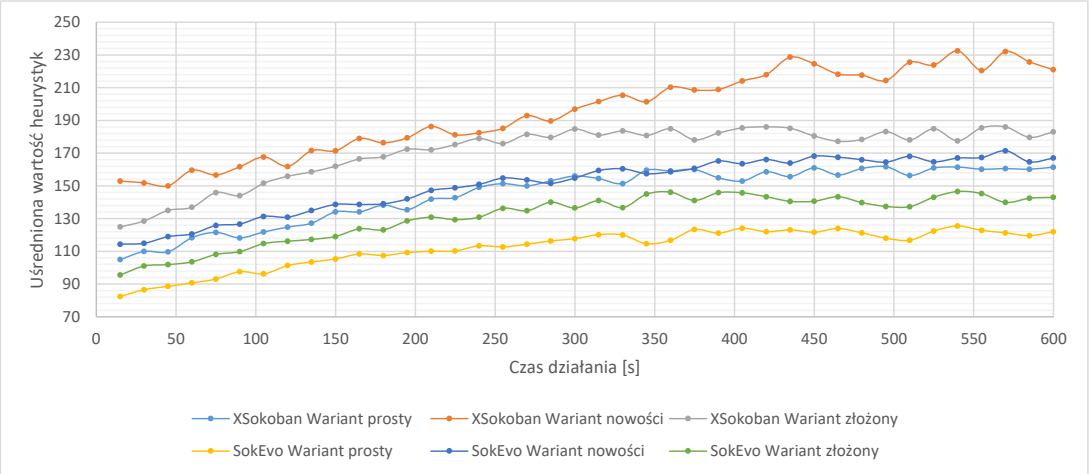
Rysunek 3.7: Zależność zużytej pamięci od czasu pracy

3.3.2. Wydajność czasowa

W celu zbadania wydajności czasowej metody PDB, postanowiono wyznaczyć uśrednione wartości heurystyk $h^{PDB_2} - h^{PDB_5}$ dla różnych czasów działania. Z reguły obserwuje się około 60% wzrost na przestrzeni pierwszych dziesięciu minut działania, potem tempo poprawy generowanych poziomów spada. Oceny poziomów ze zbioru *SokEvo* były na ogół niższe, najpewniej z uwagi na mniej skomplikowane kształty poziomów. Analizy wskazały również na to, że wariant nowości radził sobie lepiej niż wariant złożony w kwestii heurystyk baz danych wzorców.

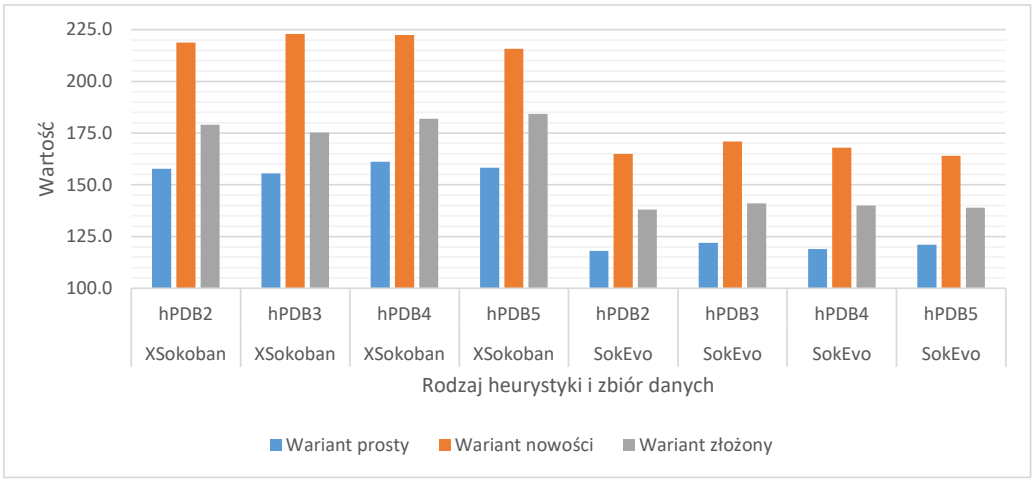
3.3.3. Funkcja nowości i liczba konfliktów

Poza wartościami heurystyk h^{PDB} należy też zbadać wartości heurystyki konfliktów kC, która również służy wycenie generowanych poziomów. W tym celu zaprezentowane zostały rys. 3.9

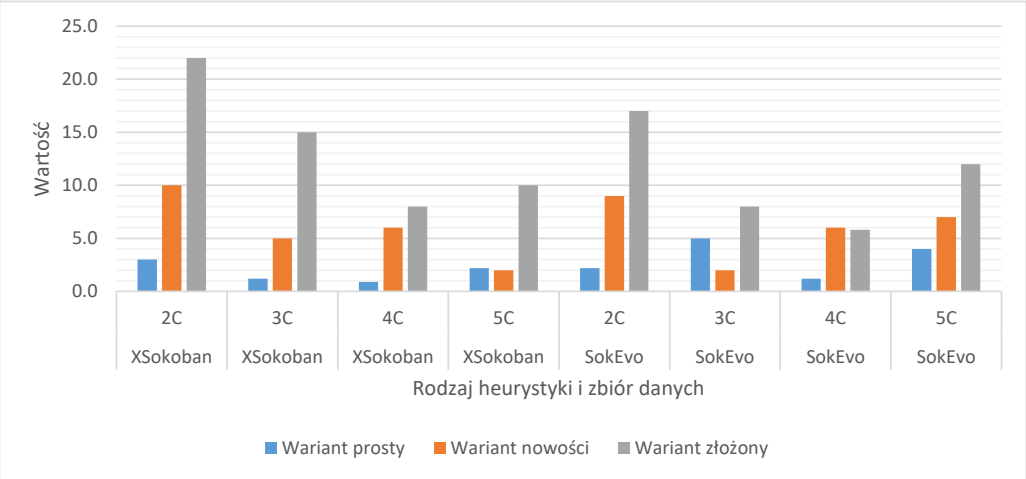


Rysunek 3.8: Zależność wartości heurystyk od czasu pracy

i 3.10, obrazujące wyniki tych badań. Jak widać, prawie w każdym przypadku wariant prosty generował plansze o najniższych wartościach heurystyk. Istotną obserwacją jest fakt, że wariant nowości generuje plansze o wyższych wartościach heurystyk baz danych wzorców, ale to wariant złożony produkuje plansze o większej liczbie konfliktów dla wzorców wszystkich analizowanych rozmiarów. Wynika to z konstrukcji sekwencji $[\cdot]$ wariantu złożonego, który uwzględnia liczbę konfliktów w trakcie przeszukiwania kolejnych stanów. Praktycznym wnioskiem, jaki należałoby wysnuć z przedstawionej analizy jest to, że wariant nowości skupia się na długości rozwiązującej ścieżki, podczas gdy wariant złożony – na liczbie zależności między pudłami.



Rysunek 3.9: Wartości heurystyk baz danych wzorców

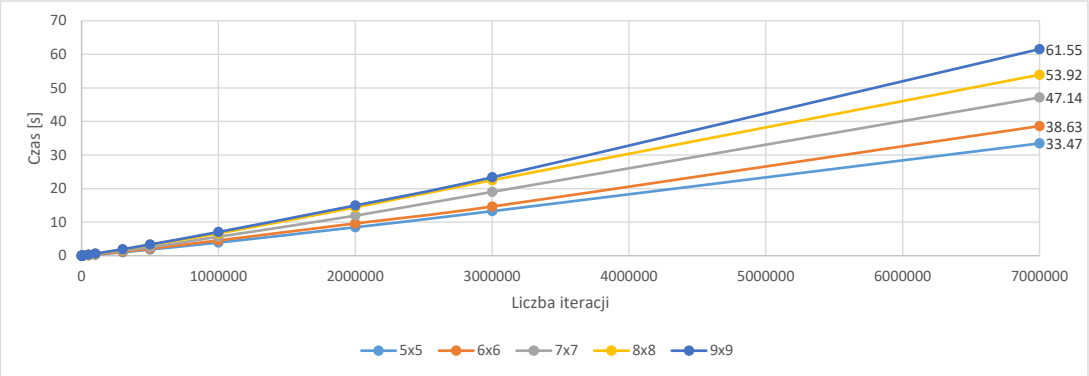


Rysunek 3.10: Wartości heurystyki konfliktów

3.4. Metoda MCTS

Wartą przytoczenia jest praca [17], w której starano się usprawnić metodę MCTS. Zaprezentowano siedem usprawnień oryginalnej metody, jednak żadne z nich nie poprawiło jej na tyle, by generowane poziomy można było uznać za zadowalająco skomplikowane.

3.4.1. Wydajność czasowa

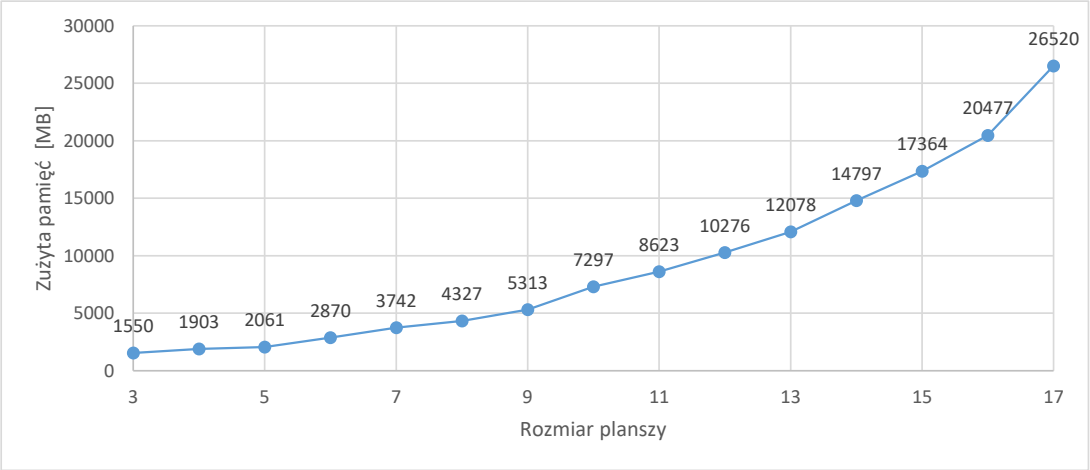


Rysunek 3.11: Zależność czasu od liczby iteracji

Wydajność czasowa metody jest najbardziej zależna od czasu rozegrania pełnej zmodyfikowanej rozgrywki. W prezentowanej implementacji liczba rozgrywek na sekundę jest zależna od rozmiaru planszy i spada wraz z ze wzrostem liczby iteracji. Dla plansz o boku krótszym niż sześć i liczby iteracji mniejszych od miliona, wykonywane jest aż do 300 000 trywialnych rozgrywek na sekundę. Ta wydajność spada prawie dziesięciokrotnie dla większych plansz i iteracji. W celu przełożenia tej wydajności na wartości czasowe, sporządzono wykres z rys. 3.11. Jak widać, dla siedmiu milionów iteracji metoda zakończy pracę po trzydziestu sekundach dla planszy o rozmiarze $5x5$, ale ta sama praca dla planszy $9x9$ zostanie wykonana po upływie minuty.

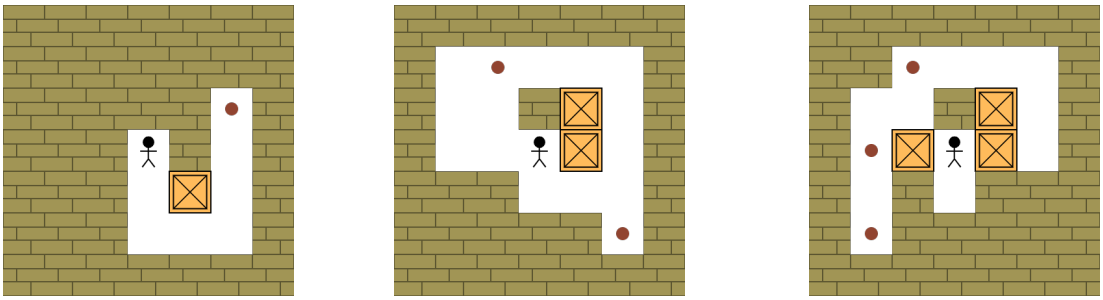
3.4.2. Wydajność pamięciowa

Zbadano ponadto zależność zużytej pamięci od liczby iteracji i maksymalnej liczby pudeł. Po przeanalizowaniu wyników eksperymentu, wnioskuje się że maksymalna liczba pudeł nie ma istotnego wpływu na zużytą pamięć. Za to wraz ze wzrostem iteracji, liniowo zwiększa się zużyta pamięć. Czynnikiem który istotniej wpływa na zużytą pamięć jest rozmiar planszy. Przeprowadzono eksperyment w którym dla 5 000 000 iteracji zbadano zużycie pamięci w zależności od rozmiaru analizowanej planszy. Jak widać na rys 3.12, ta zależność ma charakter wykładniczy i z uwagi na to, największy rozmiar planszy jaki udało się przeanalizować przy założonej liczbie iteracji to 17.



Rysunek 3.12: Zależność zużytej pamięci od rozmiaru planszy

3.4.3. Wypłaty

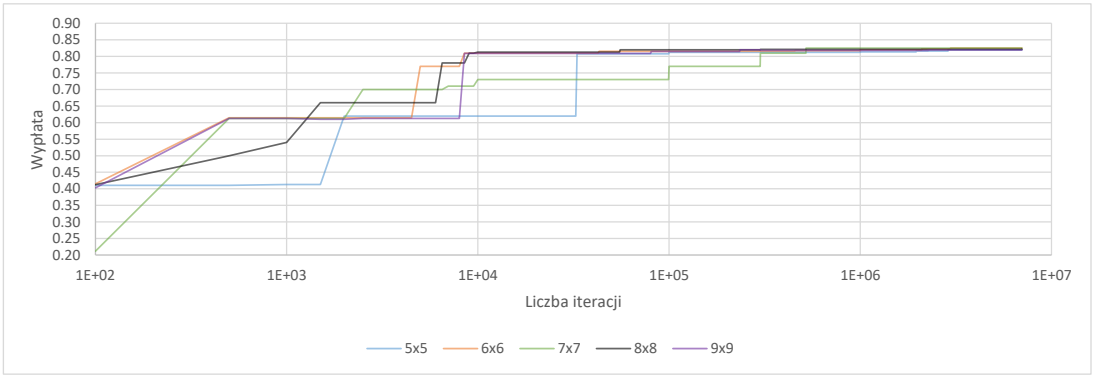


(a) Iteracje: 100, wypłata: 0.22 (b) Iteracje: 100000, wypłata: 0.62 (c) Iteracje: 10000000, wypłata: 1.1

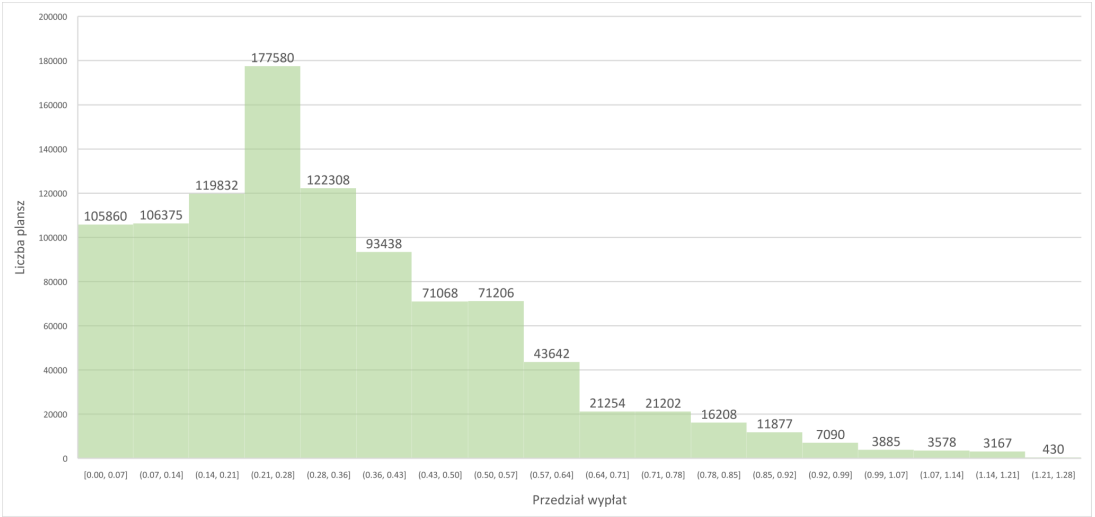
Rysunek 3.13: Rozwój generowanych plansz wraz ze wzrostem iteracji

Wartość wypłaty za wygenerowany poziom podczas wykonania metody MCTS powinna być tym wyższa, im wygenerowana plansza jest bardziej skomplikowana. Przykładowy rozwój generowanych plansz wraz z wypłatami zaprezentowano na rys. 3.13. Postanowiono zbadać, jak zmienia się najwyższa wypłata podczas kolejnych iteracji. Poznanie tej zależności pozwoliłoby na odpowiednio wczesne przerywanie działania metody, w zależności od potrzeb. Jak widać na rys. 3.14, wartość maksymalnej wypłaty jest istotnie różna dla różnych rozmiarów plansz.

Autorzy metody MCTS podają za jej zaletę fakt, iż podczas jej działania niemal co każdą iterację generowana jest pewna plansza w wyniku losowej rozgrywki. W celu zweryfikowania użyteczności sekwencji generowanych plansz, dokonano analizy średnich wypłat dla algorytmu działającego przez milion iteracji. Wyniki analiz zaprezentowano na rys. 3.15. Jak widać, niecałe 20% generowanych plansz przekracza próg wypłaty 0.5, a jedynie 0.3% z nich jest ocenione na więcej niż 1.1. Prezentowany rozkład oznacza, że jedynie wąska podgrupa z generowanych plansz może być uznana za wystarczająco skomplikowaną.



Rysunek 3.14: Zależność maksymalnej wypłaty od liczby iteracji



Rysunek 3.15: Rozkład wypłat dla generowanych plansz podczas miliona iteracji

3.5. Metoda PPO

Wszystkie eksperymenty zostały wykonane przy zachowaniu sugerowanych przez autorów [1] wartości hiperparametrów algorytmu PPO2, które wymieniono w tabeli 3.1. Wartości nagród wypłacanych agentom również bazowały na pracy [1], co wylistowano w tabeli 3.2.

Tabela 3.1: Użyte wartości hiperparametrów algorytmu PPO2

Nazwa parametru	**Wartość**
Gamma	0.99
Współczynnik entropii	0.01
Współczynnik uczenia	0.00025

Tabela 3.2: Nagrody wypłacane agentom podczas procesu treningu

Typ	**Wartość**
Poprawność poziomu	5
Obecność gracza	3
Poprawna liczba pudeł	2
Poprawna liczba pól docelowych	2
Poprawna proporcja pudeł	2
Długość rozwiązania	1

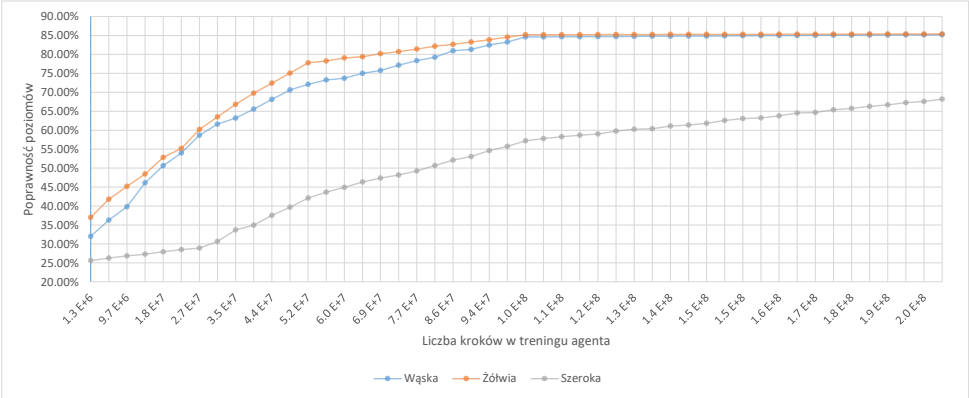
3.5.1. Poprawność plansz

Najistotniejszą różnicą między metodą PPO a pozostałymi prezentowanymi, jest ryzyko generowania niepoprawnych plansz. Przez planszę poprawną rozumie się planszę spełniającą poniższe warunki.

- Na planszy znajduje się dokładnie jeden gracz.
- Liczba pudeł odpowiada liczbie pól docelowych.
- Istnieje sekwencja ruchów gracza, która przekształca daną planszę w rozwiązaną planszę.

Wobec tego, istotne jest uczenie modeli generujących jak najwięcej poprawnych plansz, ponieważ tylko takie uznaje się za użyteczne. Zbadano zależność poprawności generowanych poziomów od liczby kroków agenta w procesie uczenia. Wyniki tych badań zobrazowano na rys. 3.16.

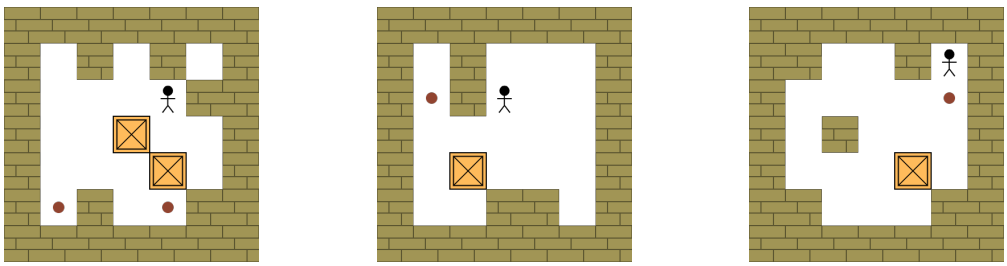
Jak widać, dla agentów opartych o reprezentację wąską i żółwią, po wykonaniu 100 000 000 kroków poprawność utrzymuje się na poziomie 80 – 85%. Taką liczbę kroków maszyna opisana w rozdziale 3.1 uzyskuje po 10 godzinach treningu. Bazując na przeprowadzonych analizach, nie warto dalej trenować agentów opartych o te reprezentacje. Jednak w przypadku reprezentacji szerokiej, między dziesiątą a dziewiętnastą godziną treningu, średnia poprawność wzrosła z 56% do 68%, zachowując wciąż tendencję rosnącą. Te różnice można tłumaczyć przez większą przestrzeń dostępnych akcji reprezentacji szerokiej. Wyniki analiz pokryły się z badaniami przedstawionymi w [1].



Rysunek 3.16: Zależność poprawności generowanych plansz od liczby kroków uczenia

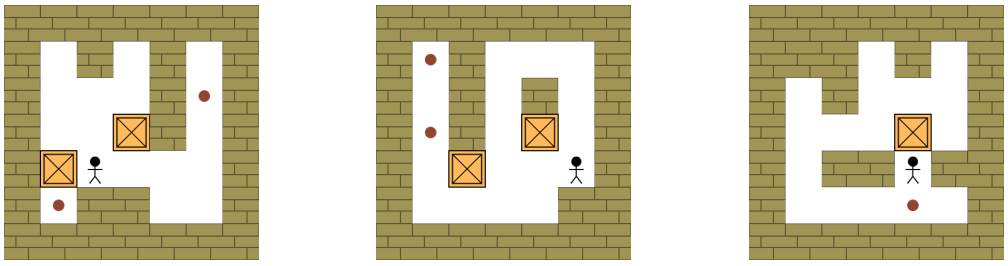
3.5.2. Wydajność czasowa treningu

Opisywane w rozdziale 3.5.1 liczby kroków uczenia należy przełożyć na czas w celu zestawiania tej metody z innymi. Liczba kroków wykonywana w jednostce czasu jest stała podczas całego procesu uczenia i wynosi dla użytej maszyny 2777 kroków na sekundę. W związku z tym, uczenie agenta przez sto milionów kroków zajmuje niecałe dziesięć godzin, a przez dwieście milionów kroków – niespełna dziewiętnaście godzin. Przykłady plansz zewaluowanych dla agentów po stu i dwustu milionach kroków uczenia zaprezentowano odpowiednio na rys. 3.17 i 3.18.



(a) Reprezentacja wąska (b) Reprezentacja żółwia (c) Reprezentacja szeroka

Rysunek 3.17: Plansze o najwyższych wypłatach dla agenta uczonego przez 9 godzin i 30 minut



(a) Reprezentacja wąska (b) Reprezentacja żółwia (c) Reprezentacja szeroka

Rysunek 3.18: Plansze o najwyższych wypłatach dla agenta uczonego przez 19 godzin

3.5.3. Wydajność czasowa ewaluacji

W rozdziale 3.5.1 zbadano zależność długości treningu od poprawności generowanych plansz, wykazując że zadowalającą poprawność (powyżej 50%) utrzymuje się po około 110 minutach treningu dla reprezentacji wąskiej i żółwiej, a dla szerokiej – po 470. Jako że proces generowanie metody PPO opiera się ewaluowaniu uprzednio wytrenowanych agentów, postanowiono zbadać również czasy ewaluacji dla trzech zwracających najlepsze wyniki w zależności do rozmiaru planszy, dokonując tysiąckrotnej ewaluacji. Jak widać w tab. 3.3, średnie czasy ewaluacji nie przekraczają trzech sekund, a dla najmniejszych plansz – jednej.

Tabela 3.3: Średnie czasy ewaluacji w sekundach

Rozmiar	**Wąska**	**Żółwia**	**Szeroka**
4	0.2	0.1	0.1
5	0.2	0.3	0.3
6	0.8	0.6	0.6
7	1.4	1.3	1.2
8	2.3	2.0	2.1
9	2.2	2.9	1.3

Mimo iż średnie czasy ewaluacji wydają się być niskie, to metoda nie daje gwarancji tak szybkiej ewaluacji, co ukazano w tab. 3.4. Maksymalne czasy ewaluacji dla większych plansz (o rozmiarach 8 i 9) przekraczały 10 sekund, w jednym przypadku sięgając nawet 19.4 s. Wyższe wartości średnie i maksymalne w przypadku reprezentacji szerokiej spowodowane są niższym skomplikowaniem generowanych plansz, które są szybciej weryfikowane przez *solver*.

Tabela 3.4: Maksymalne czasy ewaluacji w sekundach

Rozmiar	**Wąska**	**Żółwia**	**Szeroka**
4	0.2	0.2	1.1
5	1.2	3.4	0.6
6	2.2	1.6	0.7
7	8.6	5.9	5.6
8	19.4	9.6	7.3
9	15.2	12.7	10.9

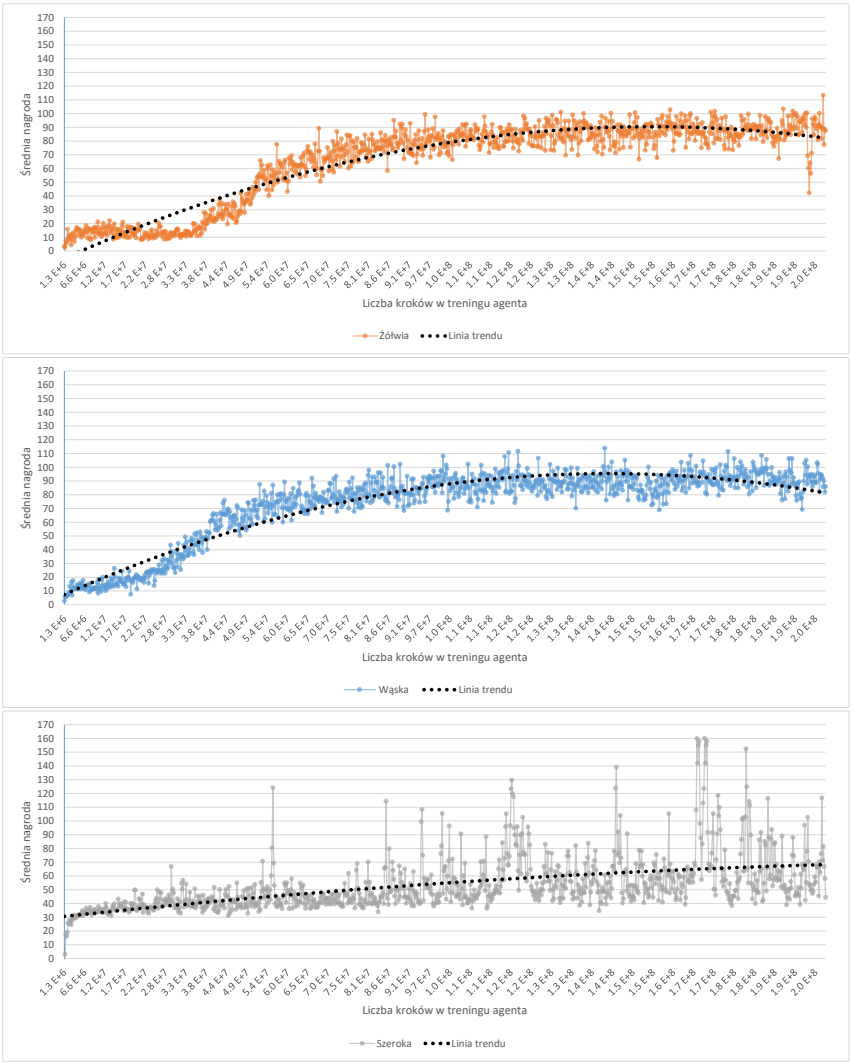
3.5.4. Nagrody

W celu zbadania tempa uczenia, przeanalizowano uśrednione wypłaty dla agentów podczas procesu treningu. Na wyniki testów ukazane na rys. 3.19, nałożono linie trendu, które potwierdzają hipotezę z rozdziału 3.5.1 o wciąż trwającym procesie poprawy agentów z reprezentacją szeroką.

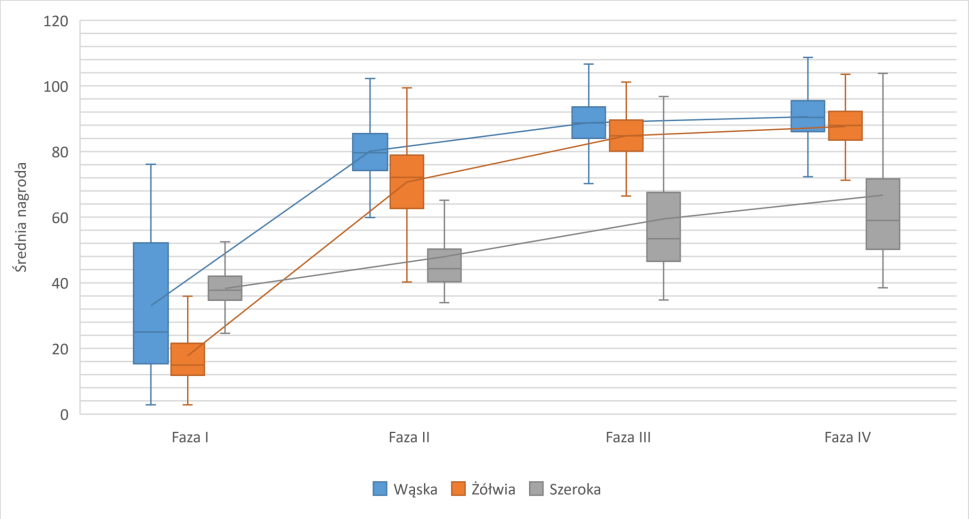
Podczas badania średnich nagród w procesie uczenia zauważono również iż ich rozkład jest zmienny w różnych etapach uczenia dla różnych reprezentacji. Po podzieleniu procesu uczenia na cztery równe fazy, stworzono wykres zmienności rozkładu średnich nagród z rys. 3.20. Jak widać, reprezentacja wąska i żółwia stabilizuje się w późniejszych fazach, zawężając tym samym rozkład otrzymywanych nagród. Reprezentacja szeroka, mimo tendencji wzrostowych, otrzymuje w każdej fazie coraz szerszy przedział średnich wypłat.

3.5.5. Rozmiar planszy

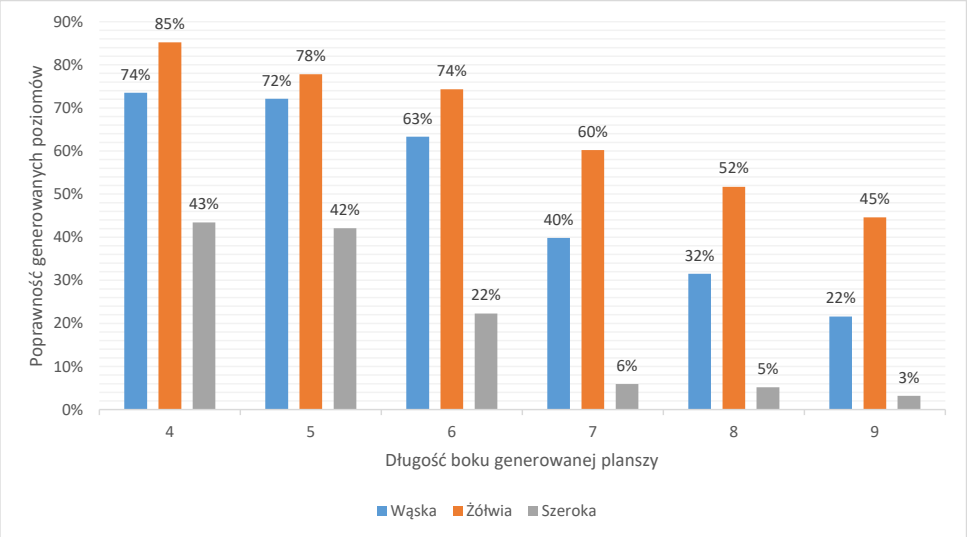
Wszystkie poprzednio opisane eksperymenty generowały plansze o rozmiarze $5x5$. Postanowiono jednak zbadać, jak metoda PPO radzi sobie z planszami większych rozmiarów. W tym celu wytrenowano dodatkowo osiemnastu agentów, z limitem na czas uczenia równym osiem godzin. Dla sześciu różnych rozmiarów plansz i trzech różnych reprezentacji dokonano później wielokrotnej ewaluacji w celu sprawdzenia średniej poprawności. Wyniki tego eksperymentu, ukazane na rys. 3.21, jasno wskazują na to, że metoda nie nadaje się dla plansz o rozmiarach większych niż $7x7$.



Rysunek 3.19: Zależność średniej nagrody od liczby kroków uczenia



Rysunek 3.20: Rozkład średnich nagród w poszczególnych fazach uczenia



Rysunek 3.21: Zależność poprawności generowanych plansz od długości boku

3.6. Ogólne metryki

Jednoznaczne wyznaczenie jakości danej planszy *Sokoban* jest problemem, nad którym pochylają się twórcy generujących je metod, przykładowo w [22] i [23]. Im lepsza metryka oceniająca planszę, tym lepsze można dać wskazówki uczącemu się algorytmowi. Każda z prezentowanych metod wprowadza pewien sposób szacowania jakości generowanych plansz, jednak sposoby te są specyficzne dla każdej metody. W celu zestawienia wszystkich metod, wprowadza się dwie autorskie metryki, opisane poniżej.

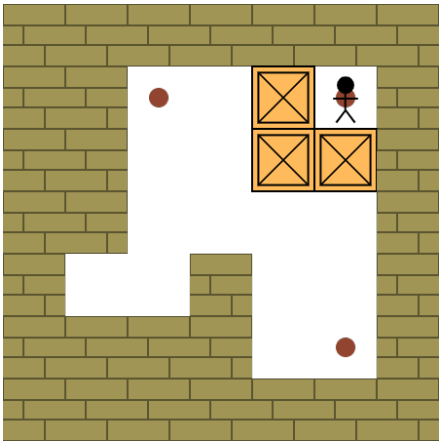
Jak wykazano w [24], ludzka ocena skomplikowania planszy *Sokoban* najbardziej pokrywa się z metrykami bazującymi na rozwiązaniu optymalnym. Jak wspomniano w p. 1.2.1, wyznaczenie jakiegokolwiek rozwiązania jest problemem *PSPACE*-zupełnym. Mimo, że metody takie jak SYM i MCTS symulują rozgrywkę, dostarczając tym samym pewne rozwiązanie planszy, to najczęściej odbiega ono istotnie od optymalnego. Wobec tego, wartości prezentowanych metryk wyznaczane są przy użyciu *solvera JSoko*.

$$PSH = \frac{p}{w + h \ + b^2} \tag{3.1}$$

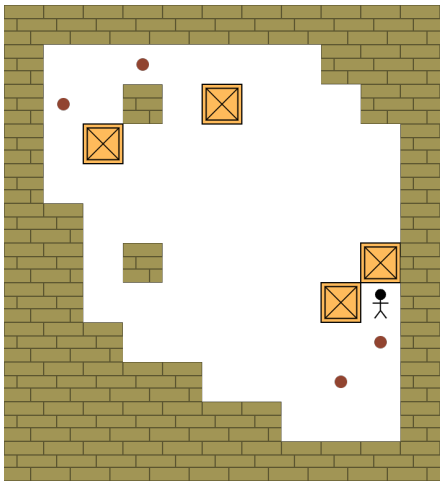
Wyznaczono dwie metryki – PSH oraz LEN, faworyzujące odpowiednio plansze wymagające dużej liczby pchnięć oraz dużej liczby ruchów gracza. Metryka opisana wyrażeniem 3.1 to iloraz minimalnej liczby pchnięć i sumy długości boków planszy oraz kwadratu liczby pudeł. Z kolei druga, z równania 3.2 to iloraz minimalnej liczby ruchów i iloczynu długości boków planszy.

$$LEN = \frac{l}{w \cdot h} \tag{3.2}$$

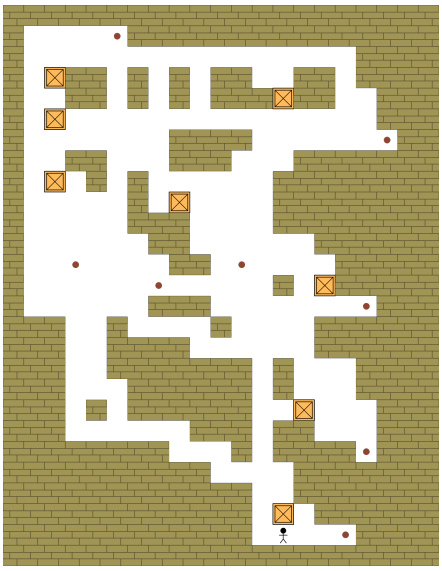
Metryki zostały zaprojektowane tak, by zwracać najwyższe wyniki dla plansz uznawanych przez ludzi za najbardziej skomplikowane. Oznacza to, że muszą być niezależne od rozmiarów plansz, co zobrazowano na rys. 3.22, gdzie wartości obu metryk nie przekraczają 0.5. Żeby zobrazować zachowanie metryk dla trudniejszych plansz, na rys. 3.23 ukazano plansze niemal trzykrotnie lepszą w ujęciu obu z nich.



(a) 5x5 - $(PSH = 0.36, LEN = 0.47)$



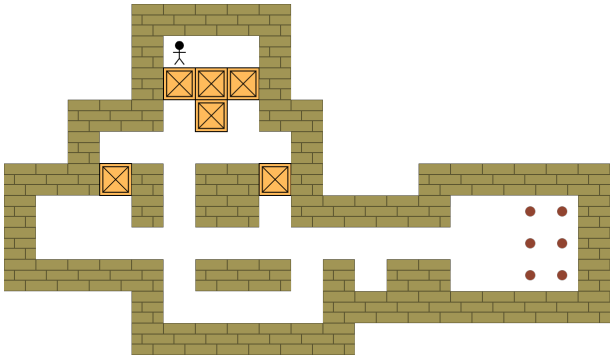
(b) 10x10 - $(PSH = 0.25, LEN = 0.32)$



(c) 20x25 - $(PSH = 0.44, LEN = 0.43)$

(d) 30x40 - $(PSH = 0.28, LEN = 0.26)$

Rysunek 3.22: Przykłady mało skomplikowanych plansz wraz z wartościami metryk PSH i LEN



Rysunek 3.23: Plansza o wartościach metryk $(PSH = 1.53, LEN = 1.38)$

3.7. Zestawienie metod

3.7.1. Wymagania sprzętowe

Każda z prezentowanych metod obarczona jest innymi ograniczeniami sprzętowymi. Metoda MCTS do swojego efektywnego działania potrzebuje operować na bardzo dużym drzewie rozgrywki, co pociąga za sobą istotne wymagania pamięciowe. Podobnie w metodzie PDB, która operuje na kolejce analizowanych stanów, wymagane jest odpowiednio dużo pamięci. W przypadku metody PPO należy wytrenować agentów przy odpowiednio dużej liczbie kroków uczenia, co jest kosztowne obliczeniowo. Metoda SYM ma najniższe wymagania sprzętowe z prezentowanych metod, ponieważ jej złożoność pamięciowa jest liniowa względem powierzchni generowanej planszy, a złożoność obliczeniowa – linowa względem zadanej liczby iteracji.

3.7.2. Skomplikowanie

Kluczową sprawą dla niniejszej pracy jest stwierdzenie, czy plansze *Sokoban* generowane proceduralnie dorównują skomplikowaniem planszom tworzonym przez ludzi. Wobec tego, postanowiono zestawić uśrednione wartości metryk obrazujących skomplikowanie dla efektów pracy analizowanych metod oraz zbiorom poziomów tworzonych przez ludzi. Jak widać w tab. 3.5, wybrano pięć zbiorów plansz i średnia wartość metryki PSH wyniosła 2.77, a metryki LEN – 7.04.

Tabela 3.5: Uśrednione wartości metryk dla zbiorów plansz

Nazwa zbioru	**Liczba plansz**	**PSH**	**LEN**
Sasquatch	50	4.72	11.92
SokHard	163	3.16	10.82
XSokoban	90	2.11	6.34
Holland	81	2.01	3.66
SokEvo	107	3.13	2.45

Zgodnie z tab. 3.6, średnie wartości metryk dla najlepszych plansz wygenerowanych przez badane metody wynoszą odpowiednio 1.6 dla PSH i 4.42 dla LEN. Wnioskuje się zatem, że średnio wypadają gorzej niż plansze tworzone przez ludzi. Należy jednak zwrócić uwagę na wysokie wartości dla metody PDB, która wygenerowała plansze bardziej skomplikowaną niż trzy z pięciu analizowanych zbiorów ludzkich. Metody SYM i MCTS generują plansze o istotnie mniejszym skomplikowaniu w metryce PSH, jednak przyglądając się problemowi od strony metryki LEN,

należy docenić najlepsze plansze metod symulujących rozgrywkę. Metoda PPO wypadła najgorzej w zestawieniu najlepszych plansz, uzyskując wyniki prawie dziesięciokrotnie gorsze w obu metrykach od metody PDB.

Tabela 3.6: Wartości metryk dla najlepszych plansz wygenerowanych przez badane metody

Metoda	**PSH**	**LEN**
PDB	3.11	9.02
SYM	1.95	4.70
MCTS	1.02	3.32
PPO	0.33	0.64

3.7.3. Wydajność czasowa

W p. 3.2.1, 3.3.2, 3.4.1, 3.5.3 zbadana została wydajność czasowa każdej z metod. Z uwagi na różne podejścia uczenia i generowania metod, nie należy zestawiać metod ze sobą. Należy zaobserwować, że mimo iż metody PPO i MCTS tworzą najwięcej poziomów w ustalonej jednostce czasu przy odpowiednim ustawieniu, większość z tych poziomów jest trywialna lub niepoprawna. Jeśli chodzi o liczbę poziomów o wartościach metryk powyżej 0.5 generowanych na minutę, wyniki takiej analizy zaprezentowano w tab. 3.7. Jak widać, metoda SYM jest zwycięzcą tego zestawienia. Metoda PDB generuje takie plansze dwukrotnie szybciej niż metoda MCTS w najpóźniejszych etapach działania. Dla metody PPO nie udało się wygenerować planszy, która osiągnęłaby przyjęte kryterium nietrywialności.

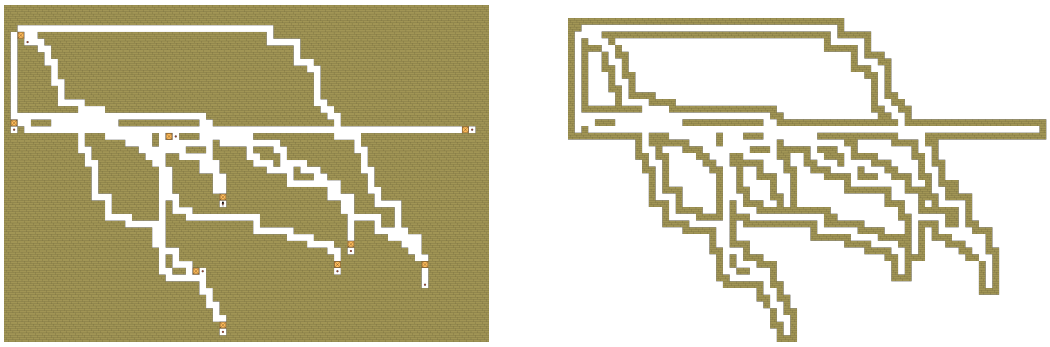
Tabela 3.7: Liczba nietrywialnych plansz generowana dla metod w minutę

Metoda	**Liczba plansz**
SYM	437.12
PDB	14.21
MCTS	6.02
PPO	0

3.7.4. Kształt i rozmiar poziomu

Przez kształt poziomu rozumie się podzbiór ścian, które są ostatnimi istotnymi polami na planszy, co zobrazowano na rys. 3.24. Istotnym faktem jest, że najbardziej skomplikowane

plansze *Sokoban* tworzone przez ludzi nie są kształtu prostokąta [22]. Z prezentowanych metod jedynie SYM i PDB mogą tworzyć takie plansze, które są zadowalająco skomplikowane. Warto jednak powtórzyć, że metoda PDB nie generuje kształtów poziomów, a jedynie wypełnia dostarczone plansze pozycjami pudeł i gracza.



Rysunek 3.24: Plansza i odpowiadający jej kształt

W kontekście rozmiaru plansz, analizy prezentowane w rozdz. 3.2–3.5 wykazały że metody heurystyczne są w stanie generować plansze o dostatecznym skomplikowaniu większe niż $6x6$. Metody oparte na uczeniu maszynowym wypadają w tej materii istotnie gorzej.

3.7.5. Podsumowanie

Jak wykazano w p. 1.2.1, problem podania rozwiązania zadanej planszy jest skomplikowany obliczeniowo i nie jest znany algorytm wielomianowy rozwiązujący go. Korzystając z metod PPO i PDB, należy wziąć pod uwagę iż nie zwracają one przykładowego rozwiązania planszy, w przeciwieństwie do SYM i MCTS, co przemawia na ich korzyść. Większość tworzonych dziś aplikacji użytkowych, skupionych na grach łamigłówkowych, prezentuje wraz ze stawianymi wyzwaniami przykładowe rozwiązania.

Warto również pochylić się nad wydajnością czasową metod. W sytuacji w której dysponuje się odpowiednio dużymi zasobami pamięci, można każdą z metod stosować równolegle, licząc na poprawę jakości wynikowych plansz. Autorzy metod MCTS i PDB prezentują właśnie taką skalowalną implementację. Metody SYM i PPO również skorzystałyby na współbieżnym uruchomieniu więcej niż jednej instancji algorytmu.

Metodę SYM cechuje możliwość tworzenia plansz o bardzo dużych rozmiarach, tym samym kreowania kształtów plansz. Metoda PDB wypada najlepiej w kwestii tworzenia skomplikowanych poziomów średnich rozmiarów, jednak wymaga gotowego schematu planszy, na którym rozmieści pudła. Metody PPO i SYM radzą sobie jedynie w tworzeniu średnio skomplikowanych plansz małych rozmiarów. W ramach głębszej analizy problemu generowania plansz

Sokoban, warto rozważyć metodę hybrydową, która mogłaby przy pomocy metody SYM generować duże poziomy, dzielić je na podproblemy rozwiązywane metodami MCTS/PPO i rozkładać finalne ułożenie pudeł przy pomocy metody PDB. Podobne podejście dzielenia problemu generacji na mniejsze części było już wykorzystywane i analizowane w literaturze [16], jednak nie korzystało ono z metod wyspecjalizowanych w danych etapach generacji.

Każda z metod opiera swoje działanie o zestaw hiperparametrów, które należy starannie dobierać w celu zmaksymalizowania jakości i ilości generowanych plansz. Analizy przeprowadzone w rozdz. 3.2–3.5 jasno wykazały, że ich dobór jest kluczowy. Wszystkie cztery opisane metody mają swoje zalety i wady, specjalizując się w różnych etapach i zadaniach generowania plansz.

Bibliografia

[1] Ahmed Khalifa, Philip Bontrager, Sam Earle, Julian Togelius, *PCGRL: Procedural Content Generation via Reinforcement Learning*, New York University, United States, 2020.

[2] Andre G Pereira, Marcus Ritt, Luciana S Buriol, *Optimal sokoban solving using pattern databases with specific domain knowledge*, Artificial Intelligence, vol. 227, pp. 52—70, 2015.

[3] Ariel Felner, Richard E. Korf, Sarit Hanan, *Additive pattern database heuristics*, Journal of Artificial Intelligence Research, vol. 22, pp.279-–318, 2004.

[4] Bilal Kartal, Nick Sohre, and Stephen J. Guy, *Data-Driven Sokoban Puzzle Generation with Monte Carlo Tree Search*, University of Minnesota, United States, 2016.

[5] Che Mat Ruzinoor, Abdul Rashid Mohamed Shariff, Biswajeet Pradhan, Mahmud Rodzi Ahmad, *A review on 3D terrain visualization of GIS data: techniques and software*, Geospatial Information Science, vol. 15:2, pp. 105–115, 2012.

[6] Damaris S. Bento, Andre G. Pereira and Levi H. S. Lelis, *Procedural Generation of Initial States of Sokoban*, Federal University of Rio Grande do Sul, Federal University of Viçosa, Brazil, 2019.

[7] Earle S., *Using fractal neural networks to play simcity 1 and conways game of life at variable scales*, AIIDE Workshop on Experimental AI in Games, 2019.

[8] Erin J. Hastings, Ratan K. Guha, Kenneth O. Stanley, *Automatic Content Generation in the Galactic Arms Race Video Game*, IEEE Transactions on Computational Intelligence and AI in Games 1, vol. 4, pp. 245–263, 2010.

[9] Florian Pommerening, Gabriele Roger, Malte Helmert, *Getting the most out of pattern databases for classical planning*, International Joint Conference on Artificial Intelligence, 2013.

[10] Francis Maes, Louis Wehenkel, Damien Ernst, *Automatic Discovery of Ranking Formulas for Playing with Multi-armed Bandits*, European Workshop on Reinforcement Learning, Athens, Greece, September 9–11, 2011.

[11] Guy Davis, *Massive - Multiple Agent Simulation System in Virtual Environment*, 2003.

[12] Jean-Yves Audibert, Remi Munos, Csaba Szepesvári, *Tuning Bandit Algorithms in Stochastic Environments*, Algorithmic Learning Theory 18th International Conference, Sendai, Japan, October 1–4, 2007.

[13] Jialin Liu, Sam Snodgrass, Ahmed Khalifa, Sebastian Risi, Georgios N. Yannakakis, Julian Togelius, *Deep learning for procedural content generation*, Neural Computing and Applications 33, pp. 19–37, 2021.

[14] Joseph C. Culberson, *Sokoban is PSPACE-complete*, 1997.

[15] Joseph C. Culberson, Jonathan Schaeffer, *Searching with pattern databases*, Canadian Conference on Artificial Intelligence, pages 402—416, 1996.

[16] Joshua Taylor, Ian Parberry, *Procedural generation of Sokoban levels*, Technical Report LARC, Laboratory for Recreational Computing, University of North Texas, 2011.

[17] Karman, S.J. *Generating Sokoban Levels that are Interesting to Play using Simulation*, Utrecht University, 2018.

[18] Levente Kocsis, Csaba Szepesvári, *Bandit based Monte-Carlo Planning*, European Conference on Machine Learning, Berlin, Germany, September 18–22, 2006.

[19] Mnih V., Kavukcuoglu K., Silver D., Rusu A. A., *Humanlevel control through deep reinforcement learning.* Nature 518(7540), 2015.

[20] Nathan Brewer, *Computerized Dungeons and Randomly Generated Worlds: From Rogue to Minecraft*, Proceedings of the IEEE, vol. 105, no. 5, pp. 970–977, May 2017.

[21] Nir Lipovetzky, Hector Geffner, *Best-first width search: Exploration and exploitation in classical planning*, AAAI Conference on Artificial Intelligence, pp. 3590—3596, 2017.

[22] Petr Jarušek, Radek Pelánek, *Difficulty rating of sokoban puzzle*, STAIRS 2010, IOS Press, pp. 140–150, 2010.

[23] Petr Jarušek, Radek Pelánek, *Human problem solving: Sokoban case study*, Faculty of Informatics Masaryk University, Brno, 2010.

[24] Petr Jarušek, Radek Pelánek, *What Determines Difficulty of Transport Puzzles?*, XXIV International FLAIRS Conference, Florida, USA, 2011.

[25] Piotr Rykała, *The growth of the gaming industry in the context of creative industries*, Uniwersytet Ekonomiczny w Katowicach, 2020.

[26] Steven James, George Konidaris, Benjamin Rosman, *An Analysis of Monte Carlo Tree Search*, University of the Witwatersrand, Johannesburg, South Africa.

[27] Vanessa Volz, Jacob Schrum, Jialin Liu, Simon M. Lucas, Adam Smith, Sebastian Risi, *Evolving Mario Levels in the Latent Space of a Deep Convolutional Generative Adversarial Network*, 2018.

[28] Walaa Baghdadi, Fawzya Shams Eddin, Rawan Al-Omari, Zeina Alhalawani, Mohammad Shaker, Noor Shaker *A Procedural Method for Automatic Generation of Spelunky Levels*, European Conference on the Applications of Evolutionary Computation, 2015.

[29] Wu Yueyang, *An Efficient Approach of Sokoban Level Generation*, Graduate School of Computer and Information Science, Hosei University, Japan, 2020.

[30] USA Today, *E3 2021: Video games are bigger business than ever, topping movies and music combined*, 2021.

[31] *Sokoban* wiki: http://www.sokobano.de/wiki.

Wykaz najważniejszych oznaczeń i skrótów

TODO - czy to jest nam w ogóle potrzebne?

Spis rysunków

Spis tabel

Spis załączników

1. Płyta CD zawierająca:
    - dokument z treścią pracy dyplomowej,
    - streszczenie w języku polskim,
    - streszczenie w języku angielskim,
    - kod programów.