Wydział Elektroniki i Technik Informacyjnych
Politechnika Warszawska

Przeszukiwanie i optymalizacja

Dokumentacja końcowa

Jan Kuc, Kacper Bugała

Warszawa, 2022Z

1. Opis problemu

Tematem projektu była implementacja i wykorzystanie algorytmu genetycznego oraz algorytmu ACO do rozwiązania *Sudoku*, a następnie porównanie ich działania przy pomocy zaprojektowanych eksperymentów numerycznych. W grze *Sudoku* dąży się do uzupełnienia planszy 9x9 (podzielonej dodatkowo na 9 mniejszych kwadratów 3x3) liczbami od 1 do 9 tak, aby te same nie powtarzały się w żadnym wierszu, kolumnie ani mniejszym kwadracie. Rozwiązanie problemu jest jednoznaczne, więc do oceny niewygrywającej planszy przygotowano funkcje celu, opisane w punkcie 2.1.6.

Całość napisana została w języku *Python*, wykorzystano bibliotekę *numpy* do optymalizacji warstwy obliczeniowej kodu. Użyta została także biblioteka *matplotlib* do wizualizacji wyników w formie wykresów. Na potrzeby zadania wygenerowano tablice sudoku przy pomocy paczki *dokusan* i dołączono je do plików źródłowych projektu, dzięki czemu nie jest wymagane zbudowanie tej paczki w środowisku lokalnym.

2. Metody rozwiązania

2.1. Algorytm ewolucyjny

2.1.1. Reprezentacja chromosomu

W zaimplementowanym algorytmie ewolucyjnym do rozwiązywania *Sudoku* reprezentacją pojedynczego chromosomu jest macierz NxN, gdzie N oznacza rozmiar planszy gry (w przypadku klasycznej wersji Sudoku N wynosi 9). W każdej z komórek macierzy wpisana zostaje wartość od 1 do 9, przy uwzględnieniu odpowiednich ograniczeń, które narzucane są przy wczytaniu planszy Sudoku do rozwiązania i generacji populacji początkowej. Macierz uzupełniana jest wierszami, zapewniając brak duplikatów w orientacji wierszowej (uwzględniane są liczby podane jako wpisane w przekazanym sudoku, patrz 2.1.2).

2.1.2. Populacja początkowa

Do wygenerowania populacji początkowej, wykorzystywana jest macierz wejściowa, opisująca początkowy stan gry (rozwiązywanie Sudoku rozpoczyna się od planszy z daną liczbą komórek wypełnionych odgórnie określoną liczbą). Macierz wejściowa, oprócz komórek z podanymi wartościami początkowymi, wypełniona jest zerami w polach 'do wpisania' przez rozwiązujacego.

Każdy chromosom generowany do populacji początkowej budowany jest według następującego schematu:

- na podstawie podanych wartości początkowych, określić zbiór możliwych wartości, które pozostały do wpisania do planszy (brakujących 1,2,3...)

- losować wartości do komórek, iterując po kolejnych wierszach macierzy, tak, aby w jednym wierszu nie było duplikatów

W dokumentacji wstępnej wspomniana została alternatywa dotycząca generowania chromosomów. Jest 'algorytm' wyglądał następująco:

- na podstawie podanych wartości początkowych, określić zbiory możliwych wartości do wpisania dla każdej z pozostałych komórek na planszy, biorąc pod uwagę kolizje w wierszach, kolumnach, ale i blokach
- w tym wypadku nie jest zwracana uwaga na duplikaty np. w wierszach

Wraz z rozwojem projektu przetestowane zostały oba podejścia, oraz zagłębiono się w dostępnej literaturze. Na wczesnym etapie nie została wykazana przewaga żadnego ze sposobów generowania populacji, wobec czego zapadła decyzja projektowa o wykonaniu testów numerycznych na algorytmie genetycznym opartym o chromosomy generowane w sposób zadany przy wywołaniu funkcji *solve* poprzez ustalenie parametru *is_candidate_mode* (candidate_mode - wybór spośród liczb-kandydatów w każdym z pól na podstawie przekazanego sudoku)

2.1.3. Selekcja

W dokumentacji początkowej planowano, aby do tworzenia kolejnych generacji chromosomów wykorzystywać selekcję turniejową. W wyniku testów i pojawiania się nowych pomysłów, zdecydowano się na zmianę selekcji.
Po pierwsze, algorytm genetyczny oparto w głównej mierze o selekcję ruletkową. Takie podejście pozwoliło na zawężanie kręgu poszukiwań wokół optimów lokalnych. Na wczesnym etapie projektu liczność optimów lokalnych i ich wpływ na znaczący wzrost trudności problemu (w szczególności dla trudniejszych plansz sudoku) został niedoszacowany. Zdecydowano się na zmianę selekcji na ruletkową, oraz zostawiono 'furtkę' na dobór hiperparametru 'succession_rate'. Ten hiperparametr wskazuje, jaka część populacji przekazywana do kolejnych epok jest wynikiem sukcesji generacyjnej - pozostałe osobniki zostaną wygenerowane w sposób losowy, tak jak przy populacji początkowej. Wpłynie to pozytywnie na zachowanie różnorodności na przestrzeni całej populacji, przy pozostaniu 'elity' w optimum lokalnym.

2.1.4. Krzyżowanie

Jako metodę krzyżowania wybrano krzyżowanie dwupunktowe, interpretowane jako zamiana określonej liczby całych wierszy między dwoma chromosomami (przy przyjętym prawdopodobieństwie krzyżowania). Takie podejście niweluje szanse na pojawienie się duplikatów w wierszach, a także na ingerencję w umiejscowienie podanych wartości początkowych gry. Odrzucono krzyżowanie jednopunktowe, ponieważ w sudoku wiersze *pierwszy* i *dziewiąty* są tak samo powiązanie, co na przykład *czwarty* i *siódmy* - różne wiersze, różne bloki. Stosowanie krzyżowania jednopunktowego 'określa' początek i koniec chromosomu, co w sudoku nie jest istotne.

2.1.5. Mutacja

Zaimplementowana mutacja zależy od ustalonego paramteru (trybu pracy) *is_candidate_mode*. W podejściu 'kandydatów' mutacja polega na zmianie wartości pola na inne, spośród liczb niewchodzących w kolizję z żadną z liczb początkowych. W podejściu 'wypełniania wierszowego', mutacja polega na przejściu przez każdy wiersz chromosomu i zamianie wartości dla dokładnie jednej pary komórek (z wyłączeniem komórek z narzuconymi wartościami początkowymi), oczywiście uwzględniając przyjęte prawdopodobieństwo mutacji. Taka mutacja również nie pozwala na pojawienie się duplikatów w rzędach, co nie jest jedynym ograniczeniem Sudoku, lecz takie założenie pozwala na lepsze ukierunkowanie przeszukiwań.

2.1.6. Funkcja celu

Funkcja celu, która służy do oceny jakości chromosomu, analizuje planszę w trzech wymiarach - wierszach, kolumnach oraz blokach. W każdej z perspektyw, znajduje i zlicza pola, których wartość nie znajduje duplikatu w swoim wierszu/w kolumnie/bloku i sumuje wynik tych trzech analiz. Maksymalny wynik jest więc równy trzykrotności liczby pól sudoku - każda komórka dostaje 'punkt' za brak kolizji w wierszu, kolumnie oraz bloku. Zadaniem algorytmu jest maksymalizacja funkcji celu, co prowadzi do znalezienia rozwiązania, czyli planszy bez duplikatów w wierszach, kolumnach ani blokach. 33

2.2. Ant Colony Optimization (ACO)

2.2.1. Reprezentacja grafu przeszukiwań

Grafową przestrzeń przeszukiwań algorytmu *ACO* dla zadania rozwiązywnia Sudoku, można sobie wyobrazić w ten sposób, że węzły to kolejne komórki planszy gry, gałęzie z nich wychodzące prowadzą do węzłów oznaczających wpisanie danej liczby do komórki, a od każdego z tych węzłów, gałąź prowadzi do kolejnej komórki planszy.
W ten sposób, każda z mrówek, ma narzucony kierunek przeszukiwań jeśli chodzi o kolejność odwiedzanych komórek.
Mrówka to jedna (z określonej liczby) instancja wykonująca algorytm - przeszukująca ścieżki grafu.

2.2.2. Cykl życia mrówki

Podczas inicjalizacji zadania, wybierana jest liczba mrówek wykonujących algorytm. Dla każdej z nich, losowany jest punkt startowy (z całej planszy gry), a następnie wykonywany jest ruch w narzuconym kierunku (np. poprzez iterowanie po kolejnych wierszach i kolumnach planszy) i wybór jednej z dostępnych wartości do danej komórki, na podstawie kosztu na określonej gałęzi i wartości feromonów "pozostawionych" przez mrówki na gałęziach w poprzednich cyklach.
W implementacji można podać dowolną liczbę mrówek, która nie przekracza liczby pól sudoku - pozycja początkowa mrówek w każdej kolejnej epoce losowana jest bez zwracania z puli wszystkich komórek w sudoku. Podróż mrówek w cyklu kończy się po odwiedzeniu przez każdą z nich wszystkich komórek planszy. W ten sposób, po zakończeniu iteracji, wybierana jest mrówka, która uzyskała najlepszą wartość funkcji celu (w tym przypadku maksymalizacja liczby komórek ze znalezionym optymalnym rozwiązaniem) i na podstawie jej trasy, aktualizowane są globalne wartości feromonów, które mają wpływ na decyzje nowych mrówek, w kolejnych cyklach. W każdym kroku po planszy, wybranie przez jedną z mrówek którejś z możliwych liczb, zmniejsza szanse innych mrówek na wybór tej samej liczby, co pozwala zapewnić szeroką rozpiętość przestrzeni przeszukiwań w każdej z epok.

2.2.3. Wykonanie ruchu

Poprawność wykonanego ruchu jest zweryfikowana zgodnie z zasadami gry w *Sudoku*. Po wczytaniu planszy początkowej z wpisanymi liczbami, dla każdej komórki wyliczane są "naiwnie poprawne" liczby, a dokładniej **niekolidujące** z wpisanymi już liczbami. Jeśli w danej komórce są dostępne jakieś liczby, wybór jednej spośród nich opiera się na losowaniu z wagami, którymi są wartości "feromonów" dla tych liczb w danym momencie. Po wylosowaniu jednej z dostępnych liczb w danej komórce (o ile taka jest), dostępne liczby dla innych komórek w tym wierszu, kolumnie oraz kwadracie zostaną zaktualizowane (wybrana liczba zostanie z nich wykluczona zgodnie z zasadami). Wylosowana liczba zostaje wpisana do indywidualnej tablicy sudoku danej mrówki, oraz pole uznane jest za optymalnie uzupełnione. Jeśli dla wybranej komórki nie ma żadnej poprawnej liczby, to pozostanie ona pusta, a komórka zostaje uznana za niepoprawną.

Po każdym uzupełnieniu komórki przez mrówkę, następuje proces propagacji ograniczeń w indywidualnej tablicy sudoku danej mrówki. Nowo wybrana liczba do konkretnego pola powoduje wykluczenie możliwości pojawienia się tej samej wartości w polach tego samego wiersza, kolumny oraz odpowiadającego bloku 3x3. Propagacja ograniczeń została zaimplementowana w klasie *Sudoku*, w metodzie *update_state()*. Każda plansza sudoku ma przy inicjalizacji ustawiany stan początkowy, co zostało zrealizowane za pomocą atrybutu struktury sudoku o nazwie *state*. Jest to słownik przechowujący zbiór dostępnych liczb do wpisania do każdej z niewypełnionych, przy definicji planszy, komórek. Propagacja ograniczeń jest mechanizmem rekurencyjnym, którego zadaniem jest aktualizacja dostępności numerów do wpisania dla każdego z wolnych pól oraz ustawianie ich jako "fixed", jeśli nie ma żadnej innej opcji, lub "failed", gdy nic nie zostało jeszcze wpisane do komórki, a nie ma już dostępnych liczb na to miejsce. Rekurencja została zastosowana po to, by przy wprowadzaniu ograniczeń wynikających z wpisania liczby do aktualnego pola mrówki, uwzględnić też ewentualne kolejne ograniczenia, które mogą się pojawić w trakcie tego procesu.

3. Opis eksperymentów numerycznych

Przeprowadzono liczne eksperymenty numeryczne pod kątem badania jakości zaimplementowanych algorytmów. Analizowano także wpływ zmian wartości hiperparametrów. Wszystkie testy przeprowadzono na dużej próbie, a wyniki uśredniano, co pozwoliło zminimalizować błędy statystyczne (oba algorytmy są niedeterministyczne). Zapisane zostały ziarna generatorów liczb pseudolosowych, których wykorzystanie pozwala na reprodukcję wszystkich przeprowadzonych eksperymentów numerycznych.

Testy podzielono na trzy poziomy zaawansowania Sudoku ('easy', 'medium' oraz 'hard), co pozwoli na przeanalizowanie rozwiązań na różnym poziomie trudności problemu. Algorytymy badano pod kątem:
— złożoności obliczeniowej,
— przyrost wartości funkcji celu,
— skuteczność.

Złożoność obliczeniowa jest jednym z głównych wskaźników jakości w ocenie implementacji algorytmu. Ze względu na znaczące różnice pomiędzy dwoma algorytmami, ocena złożoności jako liczbie wykonanych iteracji byłaby niepoprawna – w obu implementacjach jedna epoka odpowiada za co innego, przez to złożoność jednej epoki nie jest równa drugiej. Złożoność została zbadana w formie czasu potrzebnego do wykonania skryptu. Takie rozwiązanie pozwala spojrzeć obiektywnym okiem na złożoność obliczeniową obu algortymów.
Przyrost wartości funkcji celu zbadany został w formie analizy najlepszej wartości uzyskanej do i-tej epoki. Tutaj podobnie jak w poprzednim przypadku – epoka epoce nierówna. Dane przedstawiane są w formie wykresów, z których odczytać można **wartość oczekiwaną** na tle **rozkładu wyników** z przeanalizowanej próby, rozumianej jako rozpiętość między najlepszą a najgorszą symulacją w próbie. W połączeniu z odczytanym czasem propagacji pozwala na wyciągnięcie bardzo istotnych wniosków.
Skuteczność to najbardziej trywialny wskaźnik jakości – oceniane jest rozwiązanie sudoku w ograniczonej liczbie iteracji w sposób binarny. **Liczba epok**, której potrzebował algorytm na rozwiązanie sudoku (lub maksymalna liczba epok, w przypadku, w którym algorytm nie znalazł ostatecznego rozwiązania). Stanowi dobry wskaźnik tego jak szybko algorytm zbiega do oczekiwanego rozwiązania. Jest kluczową miarą decydującą o tym, jakie nastawy danego algorytmu dają rozwiązanie najszybciej.

Hiperparametry:

— Algorytm genetyczny:
1. *pm* - prawdopodobieństwo mutacji (w zależności od 'trybu')
2. *pc* - prawdopodobieństwo krzyżowania dwupunktowego
3. *pop_size* - rozmiar populacji
4. *max_epoch* - liczba epok, po której zadanie zostaje uznane za nierozwiązane
5. *reset_condition* - liczba epok, po których resetowana jest populacja w przypadku braku postępów wartości funkcji celu
6. *succession_rate* - część populacji będąca wynikem sukcesji generacyjnej (ruletkowej) z poprzedniej iteracji
7. *if_candidates* - wybór sposobu generowania chromosomów, losowanie spośród początkowo niezakazanych wartości, lub wypełnianie sudoku wierszami

— Ant Colony Optimization:
1. *ants_count* - liczebność mrówek w pojedynczej epoce
2. *greed_factor* - współczynnik *chciwości*, który określa jak często wybierana będzie najlepsza dotychczasowa decyzja (zgodnie z tablicą feromonów)
3. *max_epoch* - liczba epok, po której zadanie zostaje uznane za nierozwiązane
4. *evaporation* - współczynnik zanikania wartości feromonów
5. *global_pher_factor* - współczynnik wpływający na adaptację wartości w globalnej tablicy feromonów kolonii
6. *local_pher_factor* - współczynnik wpływający na adaptację wartości w lokalnej tablicy feromonów mrówki

4. Wyniki eksperymentów

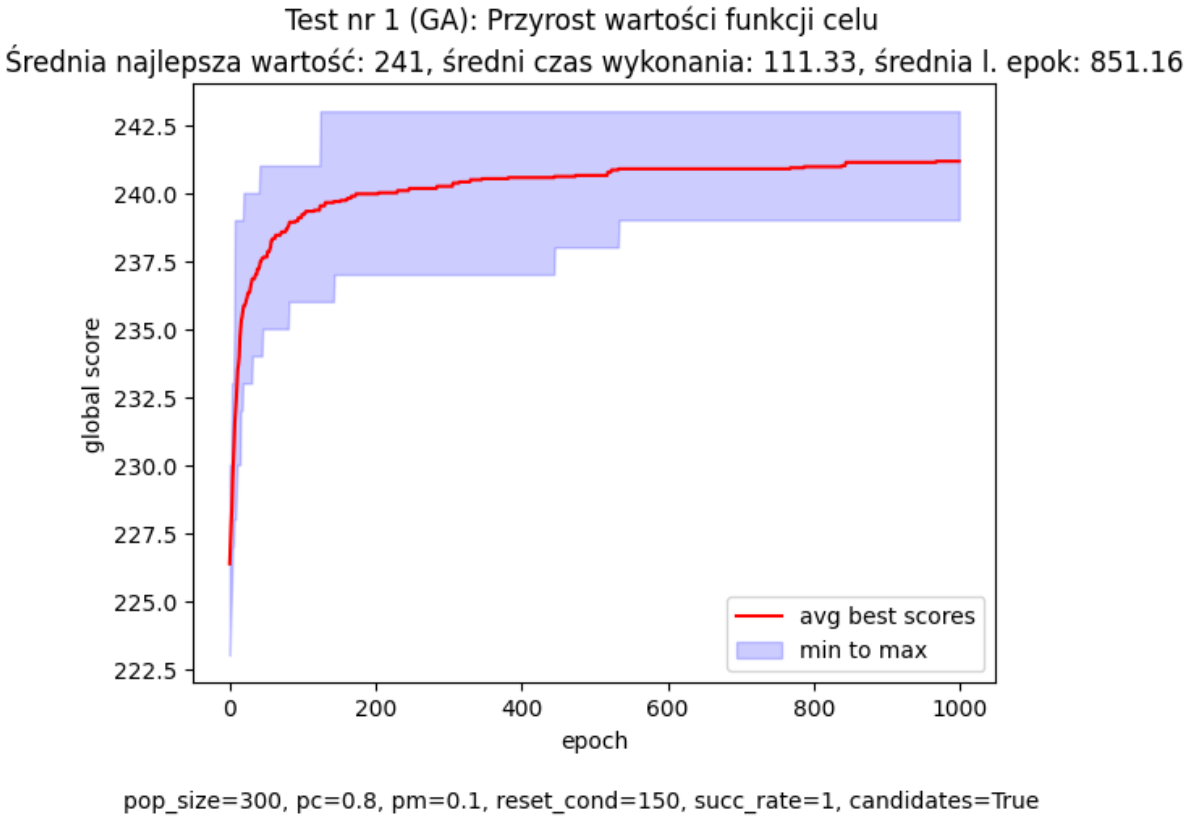
4.1. Algorytm genetyczny

Wykresy przyrostu oczekiwanej wartości funkcji celu zaprezentowano w tej sekcji. Tak jak opisano w 2.2.3, w przypadku algorytmu genetycznego maksymalną wartością funkcji celu jest wartość 243 (unikatowość każdej z liczb w swoim bloku, wierszu i kolumnie). Epizody zakończone tym wynikiem uznaje się za rozwiązane. Średnia liczba epok oraz czas wykonania zapisano w przypisie wykresów.

4.1.1. Sudoku - poziom łatwy

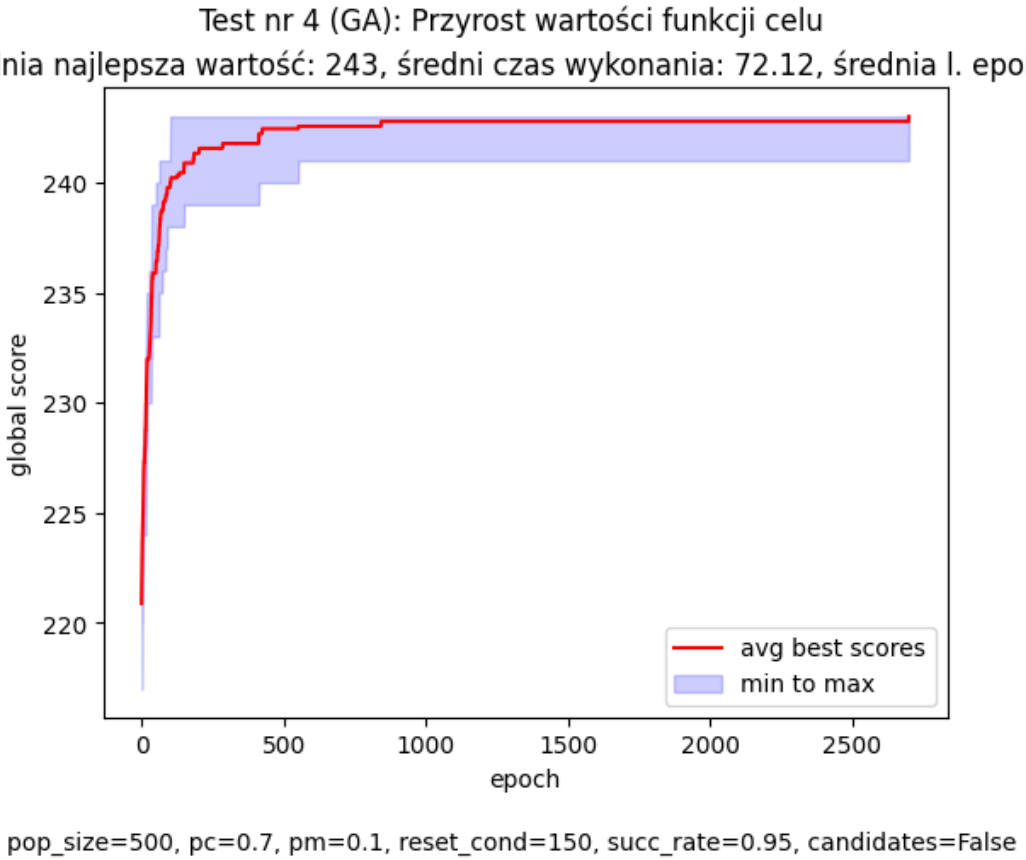
Wyniki jednego z pierwszych testów zaprezentowano na wykresie 4.1. Został on uruchomiony w trybie generowania chromosomów z ujęciem "kandydatów" na pole. Średnia wartość po 1000 epokach osiągnęła wartość 241. Oznacza to, że nie wszystkie plansze zostały rozwiązane. 1000 epok nie wystarczyło, aby dla tych wartości parametrów algorytm rozwiązywał wszystkie plansze w tej liczbie iteracji. Średni czas wykonania wyniósł 111 sekund, co daje niemal 2 minuty – nie jest to zadowalający wynik. W najlepszych uruchomieniach sudoku zostało rozwiązane już po 180 epokach. Najgorszy uzyskany wynik to 239 - czyli bardzo blisko rozwiązania. Można domniemywać, że z perspektywy funkcji celu to jedno lub dwa pola generujące kolizje. Znajomość problemu sudoku pozwala zakładać, że algorytm utknął w optimum lokalnym, który mimo wysokiej wartości funkcji celu jest bardzo odległy od optymalnego, jednoznacznego rozwiązania. Uruchomienie algorytmu na większym limicie maksymalnych iteracji z pewnością pomogłoby w

znalezieniu rozwiązania, lecz czas krytyczny został już osiągnięty - wartości parametrów można uznać za niekorzystne.



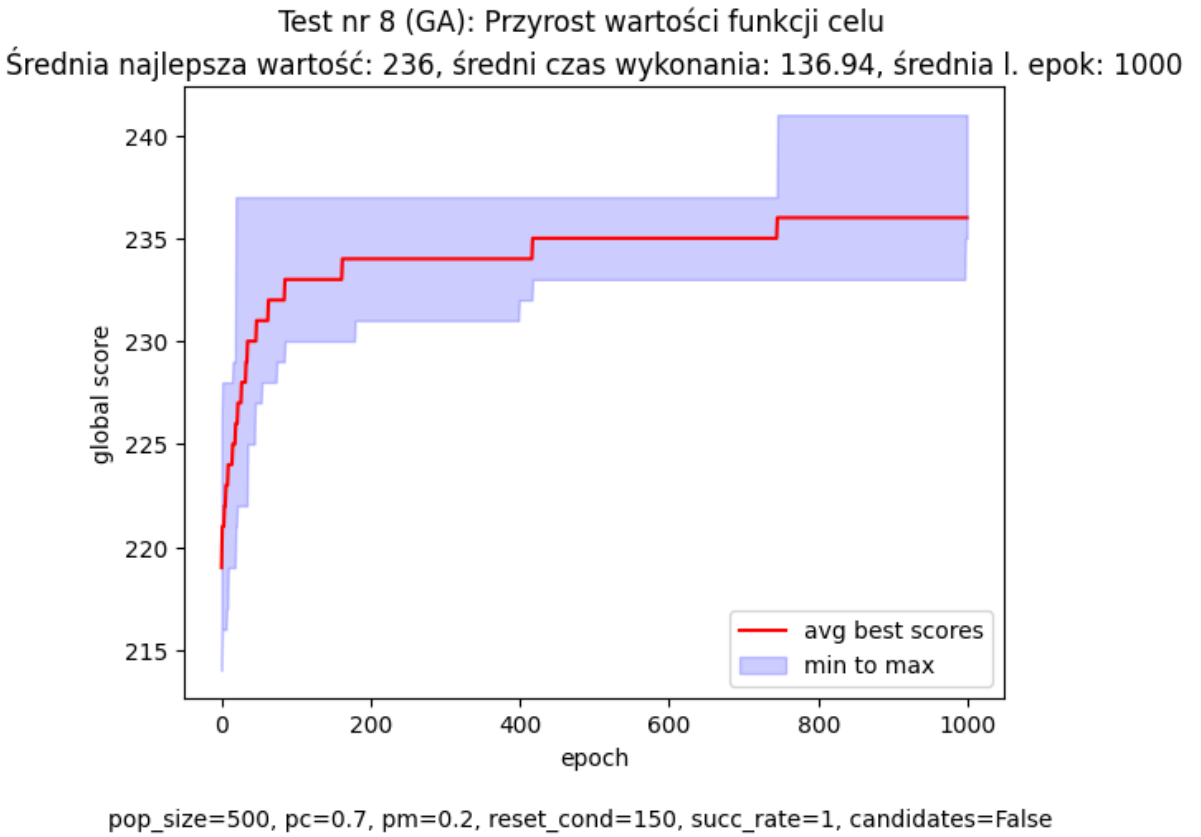
Rys. 4.1. Test nr 1 algorytmu genetycznego

Test nr 4 (rys. 4.2) przyniósł znacznie lepsze wyniki. Zdecydowano się na zmianę podejścia generacji kandydatów na metodę "wierszową", oraz dorzucono 5% zupełnie losowych osobników w każdej epoce (*succession_rate*). Populacja zwiększona została do 500 osobników, a szansę na krzyżowanie zmniejszono. Algorytm testowano na znacznie zwiększonej maksymalnej liczbie epok, co jednak nie przełożyło się negatywnie na średni czas propagacji. Każda z plansz rozwiązywana była średnio w 72 sekundy. Jedna z plansz została rozwiązana już po 200 epokach, po 500 epokach większość była już rozwiązana. Ostatecznie po 3000 epokach wszystkie planszy były już rozwiązane. Wykres 'średniej' pokazuje oczekiwaną liczbę epok do rozwiązania sudoku na poziome 700. Takie wartości hiperparametrów zostały uznane za bardzo dobre.



Rys. 4.2. Test nr 4 algorytmu genetycznego

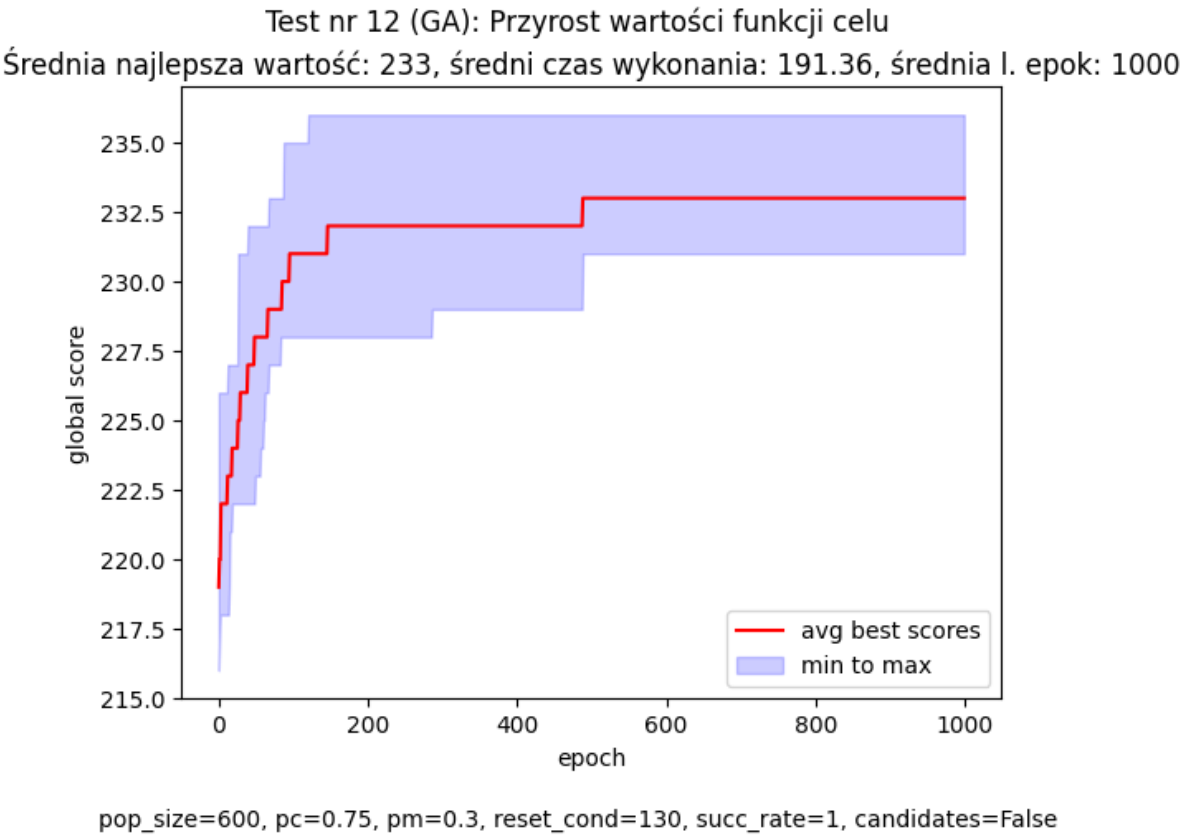
Test nr 8 (rys. 4.3) pokazał istotność implementacji selekcji niecałkowicie sukcesywnej. W tym eksperymencie zdecydowano się na wykorzystanie tylko i wyłącznie sukcesji generacyjnej do selekcji osobników w kolejnych epokach. Rekompensatą za usunięcie losowych, nowych osobników (krok w kierunku eksploracji przestrzeni) zostało zwiększenie prawdopodobieństwa mutacji. Wykres jasno pokazuje, w szczególności w porównaniu z testem nr 4 (rys. 4.2, że wyniki są słabe. Duży rozrzut jakości, bardzo niski stopień rozwiązanych plansz. Przeprowadzono także testy bez zwiększenia wartości współczynnika mutacji, co dało te same efekty.



Rys. 4.3. Test nr 8 algorytmu genetycznego

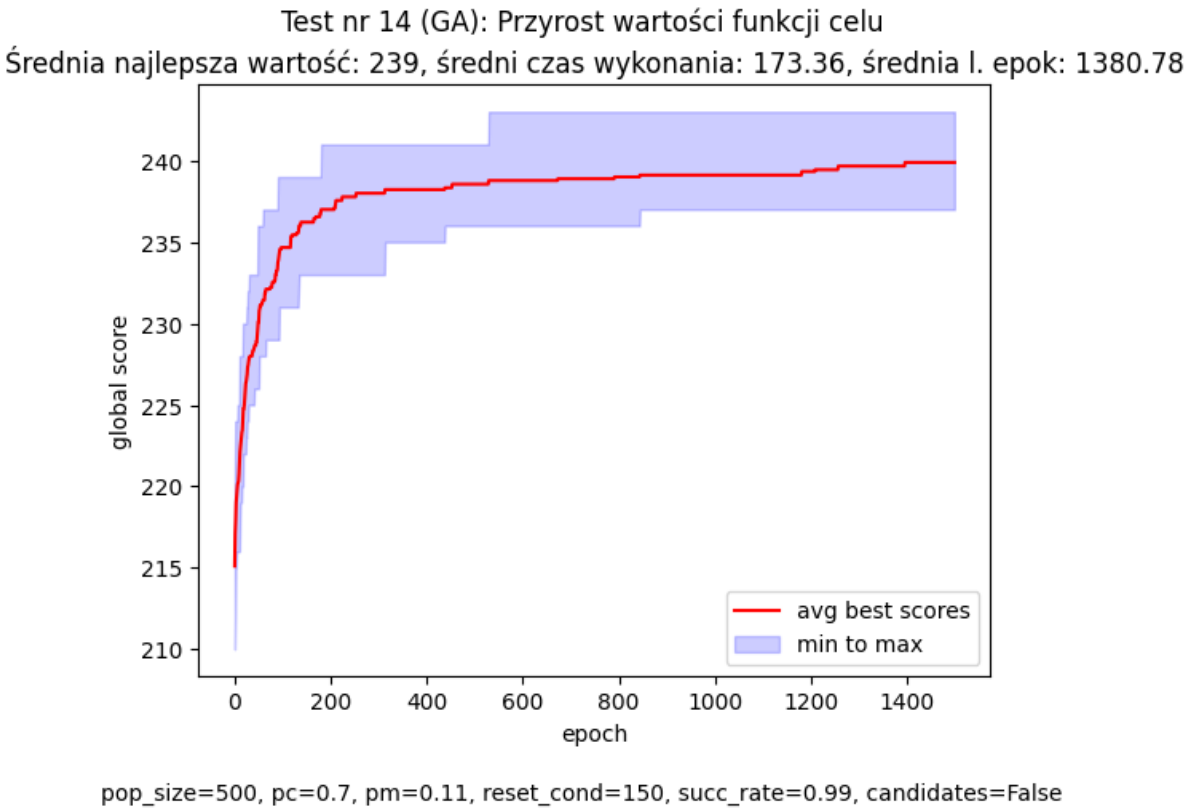
4.1.2. Sudoku - poziom średni

Algorytm genetyczny testowano również na trudniejszych planszach. Już na planszach łatwych algorytm nie działał idealnie, wnioskować więc można że na tych o wyższym poziomie trudności (więcej pól do zapełnienia) algorytm nie będzie działać lepiej. Eksperyment 12 (rys. 4.4) to szybko potwierdził. Średni wynik po tysiącu iteracji wyniósł jedynie 233. Bardzo niska jakość algorytmu. Konsekwencją tego jest też 191 sekund wykonywania każdego z epizodów, co jest wartością drastycznie wysoką.



Rys. 4.4. Test nr 12 algorytmu genetycznego

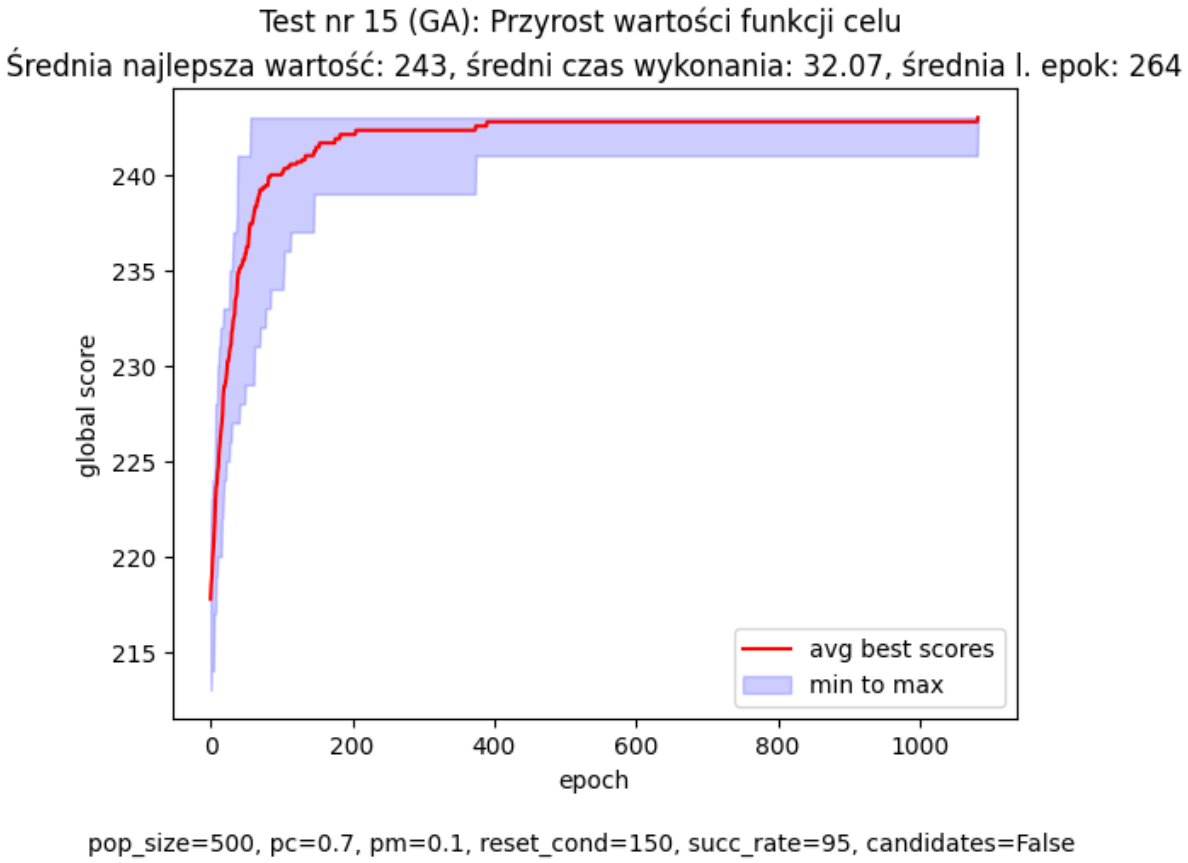
Przetestowano też najlepsze wartości ustawień znalezione dla plansz łatwych, oraz wartości podobne do nich, które w większości przypadków zwracały wyniki wysokiej jakości. Udało się rozwiązać większość zadań w optymalnym czasie. Jednym z mniej udanych eksperymentów był test nr 14 (rys. 4.5), który przy *succession rate=0.99* nie dał żadnego poprawnego rozwiązania sudoku



Rys. 4.5. Test nr 14 algorytmu genetycznego

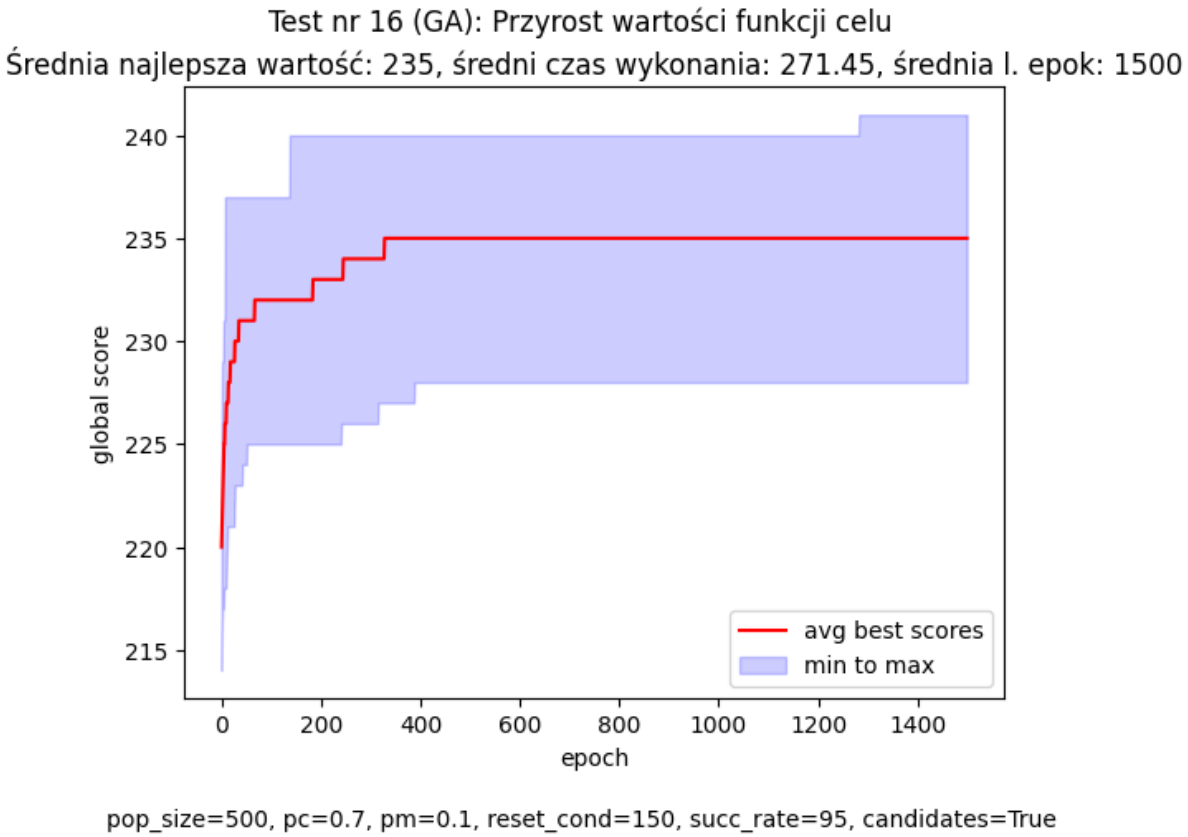
4.1.3. Sudoku - poziom trudny

Eksperymenty numeryczne algorytmu genetycznego na planszach najtrudniejszych zaczęto od przetestowania rozpoznanych wcześniej najlepszych wartościach parametrów, przy użyciu "wierszowej" generacji chromosomów. Wynik testu (rys. 4.6) kolejny raz potwierdził wysoką jakość tych ustawień, rozwiązując wszystkie zadane plansze w ciągu maksymalnie 1000 epok. Najlepsze epizody zakończono już w setnej epoce, co jest świetnym wynikiem na tle pozostałych ustawień dla tej skali problemu. Średni uzyskany czas propagacji wyniósł 32 sekundy.



Rys. 4.6. Test nr 15 algorytmu genetycznego

Dla ostatecznego porównania dwóch podejść generacji chromosomów wraz z ich mutacją przeprowadzono eksperyment na tych samych planszach co w eksperymencie 4.6. Wyniki jednoznacznie i ostatecznie wskazują, że to podejście wierszowe, wraz z mutacją pojedynczych pól w obrębie wiersza daje najlepsze wyniki. Pozwala to znaleźć rozwiązania nawet najtrudniejszych plansz, można uznać je też za zdecydowanie powtarzalne, a kowariancja wyników pozostaje w normie. Podejście "kandydatów" nie zwróciło żadnego poprawnego rozwiązania, mimo wyczerpania maksymalnej liczby iteracji w każdym z epizodów - algorytm z tymi ustawieniami jest całkowicie nieskuteczny.



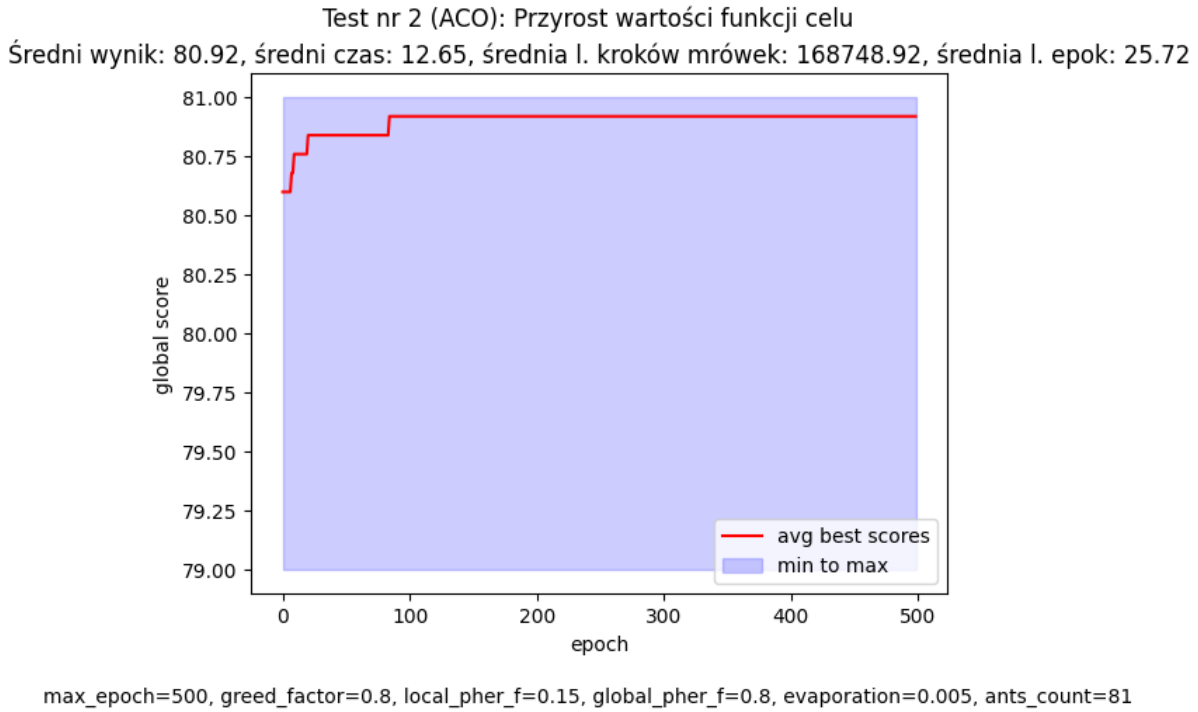
Rys. 4.7. Test nr 16 algorytmu genetycznego

4.2. ACO

W algorytmie Ant Colony Optimization funkcja celu osiąga wartość maksymalną równą 81. W tym przypadku również badamy czas propagacji programu, lecz w tym przypadku złożoność pojedynczej epoki zależy od jednego z hiperparametrów - ants_count. Oznacza on liczbę mrówke poruszających się po przestrzeni stanów w każdej z iteracji. Dla porównywania złożoności obliczeniowej dla różnych ustawień, zliczana jest średnia liczba kroków mrówek, gdzie każda mrówka wykonuje 81 kroków w każdej iteracji. Takie podejście jest bardzo rozsądne, ponieważ każdy ruch każdej z mrówek idzie w parze z propagacją ograniczeń na jej własnej tablicy sudoku. Zliczanie ruchów mrówek pozwoli oszacować liczbę wykonanych operacji na zadanej tablicy sudoku, po której algorytm jest w stanie wskazać poprawne rozwiązanie.

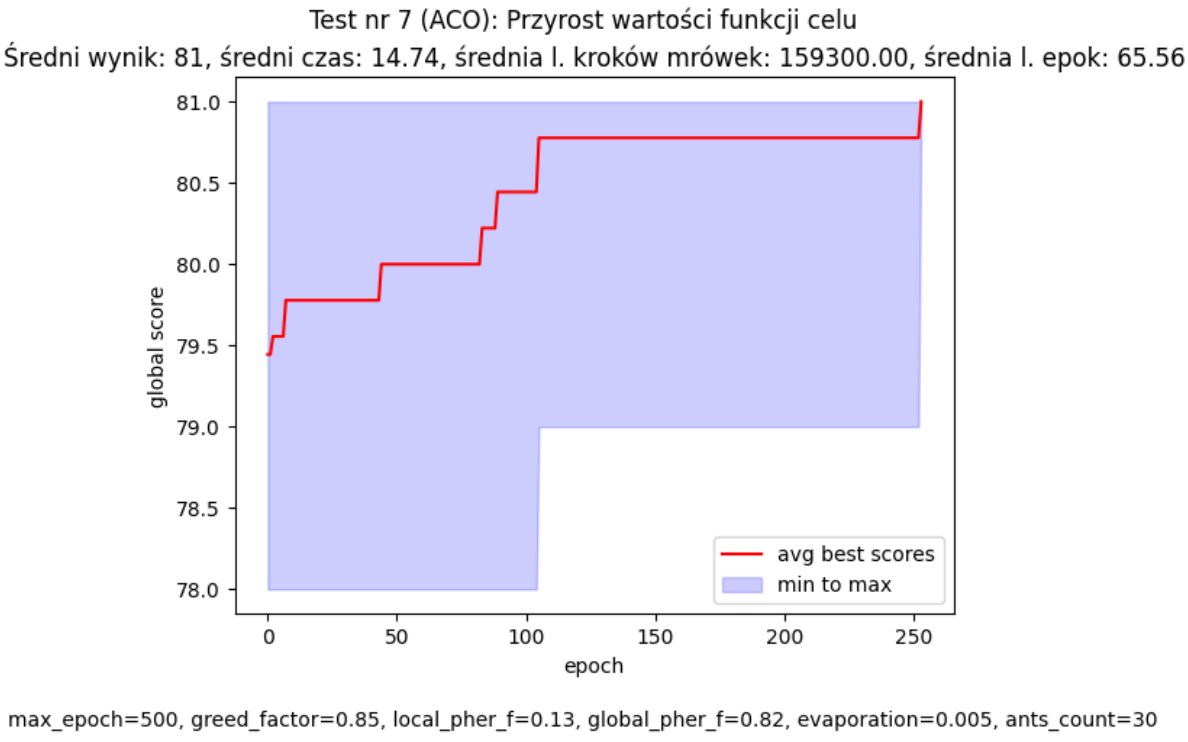
4.2.1. Sudoku - poziom łatwy

Pierwsze eksperymenty numeryczne przeprowadzone na algorytmie ACO pokazały potencjał tego algorytmu. Podczas gdy algorytm genetyczny potrzebował często ponad minuty na osiągnięcie jakichkolwiek rozwiązań planszy, to ACO potrafiło znaleźć rozwiązanie po kilku sekundach. Początkowo założone ograniczenie 500 iteracji okazało się trwać bardzo krótko. Wykres pokazuje, że niektóre epizody rozwiązane zostały już w pierwszej epoce. Najgorszy wynik po pierwszej epoce wyniósł 79, co oznacza jedynie dwie kolizje na całej planszy. Jest to bardzo dobry wynik. Przeprowadzono też kolejne testy na sudoku łatwym, w celu zbadania wpływu zmiany parametrów na jakość algorytmu.



Rys. 4.8. Test nr 2 algorytmu ACO

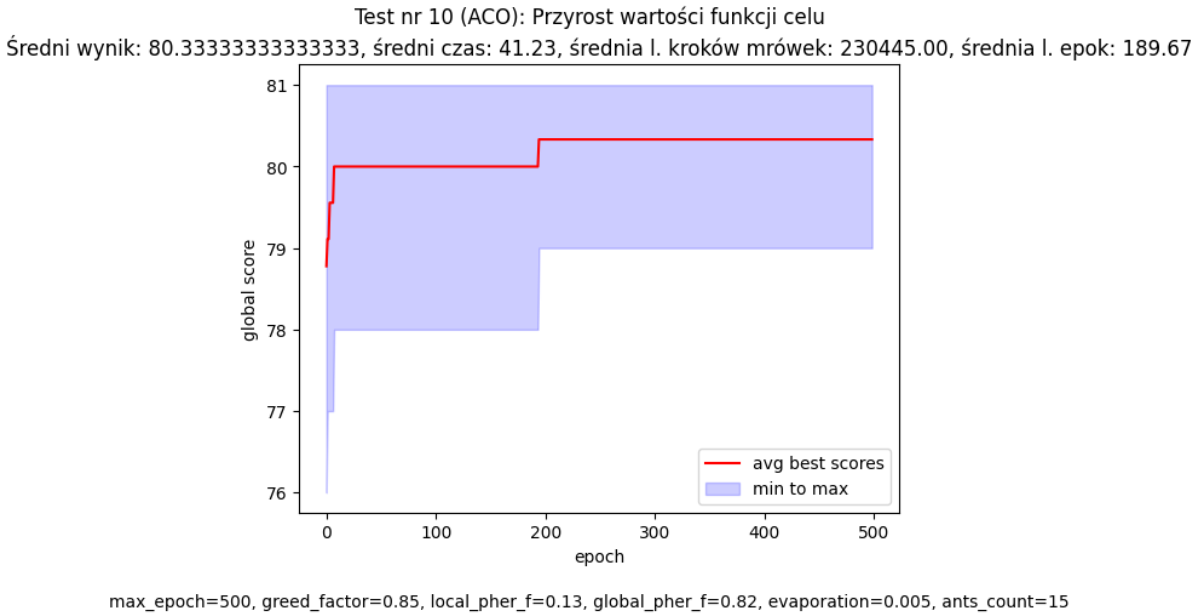
Wyniki testu nr 7 (rys. 4.9) świadczą o istotności parametru "chciwości" (greed_factor). Tutaj nieznaczne zwiększenie jego wartości rozwiązało wszystkie plansze już po 250 iteracjach (średni wynik = 81, czyli plansza rozwiązana). Test ten został przeprowadzony z wynikiem średnej liczby kroków mrówek równej 160000. jest to wynik zdecydowanie niższy, niż w chociażby zaprezentowanym teście nr 2 (rys. 4.8).



Rys. 4.9. Test nr 7 algorytmu ACO

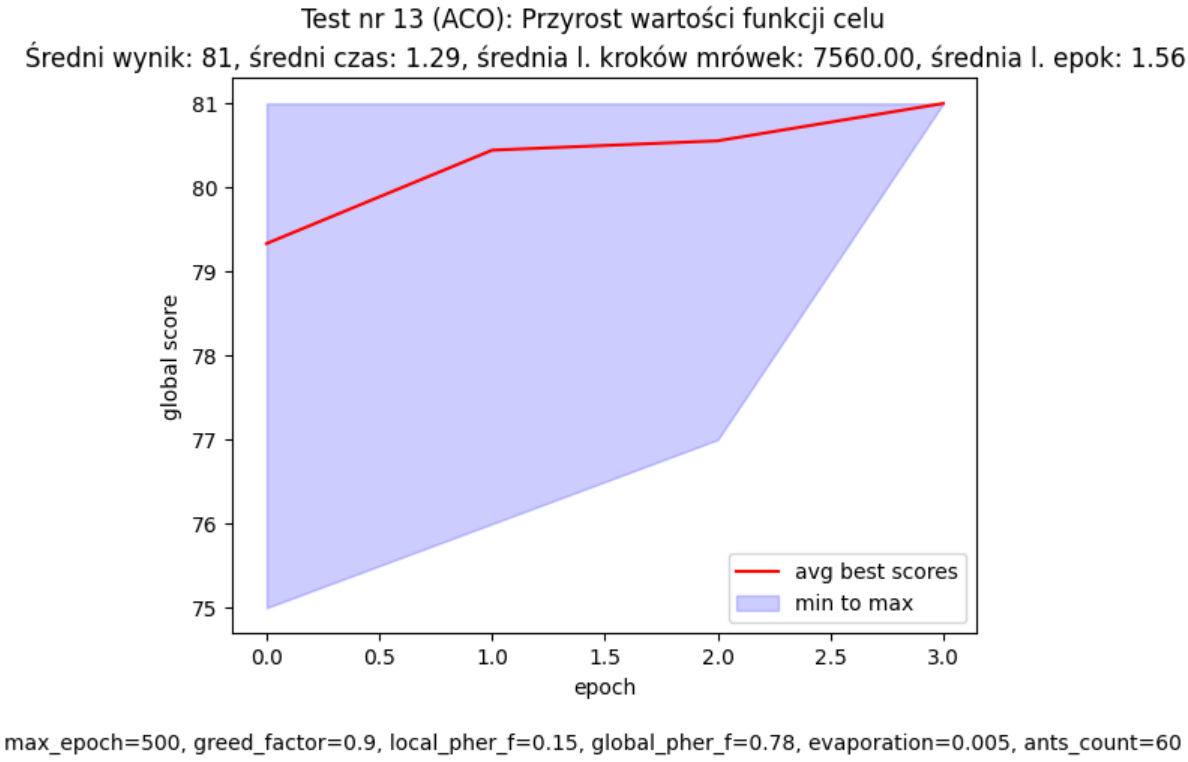
4.2.2. Sudoku - poziom średni

Wbrew spodziewanym wynikom, algorytm dla teoretycznie trudniejszych plansz uzyskiwał podobną skuteczność. Może to wynikać m.in. z poprawnego doboru parametrów wyznaczonych na planszach łatwiejszych. Test nr 10 (rys. 4.10) przyniósł większy czas propagacji i większą liczbę kroków, lecz wysoką i zadowalającą skuteczność po 500 iteracjach (na poziomie 50%).



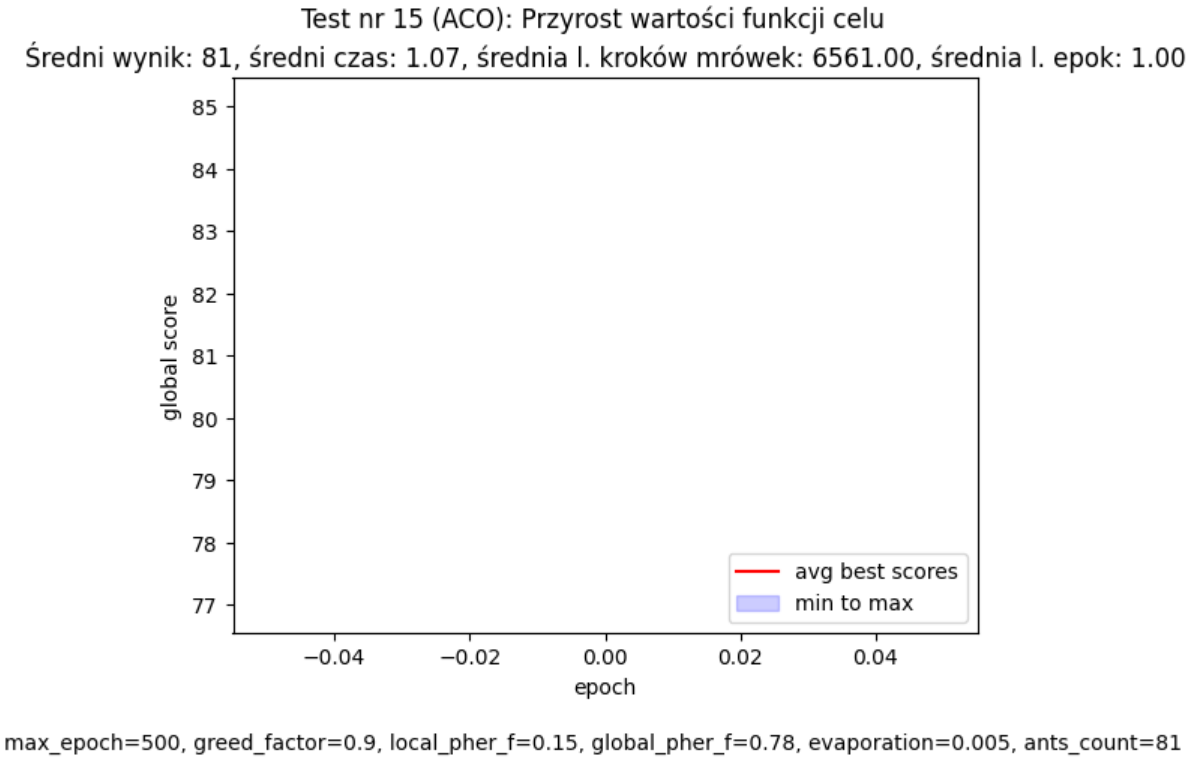
Rys. 4.10. Test nr 10 algorytmu ACO

W teście 13 (rys. 4.11) postawiono na zwiększenie liczby mrówek. Ustalenie ich liczności na 60 oraz chciwości na 0.9 sprawiło, że każde z zadanych sudoku niemal od razu zostały rozwiązane - w najgorszym przypadku algorytm potrzebował 3 iteracji. Wartość oczekiwana czasu znalezienia rozwiązania wyniosła 1.29 sekundy. Kolejne eksperymenty pokazują, że chciwość i liczność mrówek to najważniejsze hiperparametry metody mrówkowej.



Rys. 4.11. Test nr 13 algorytmu ACO

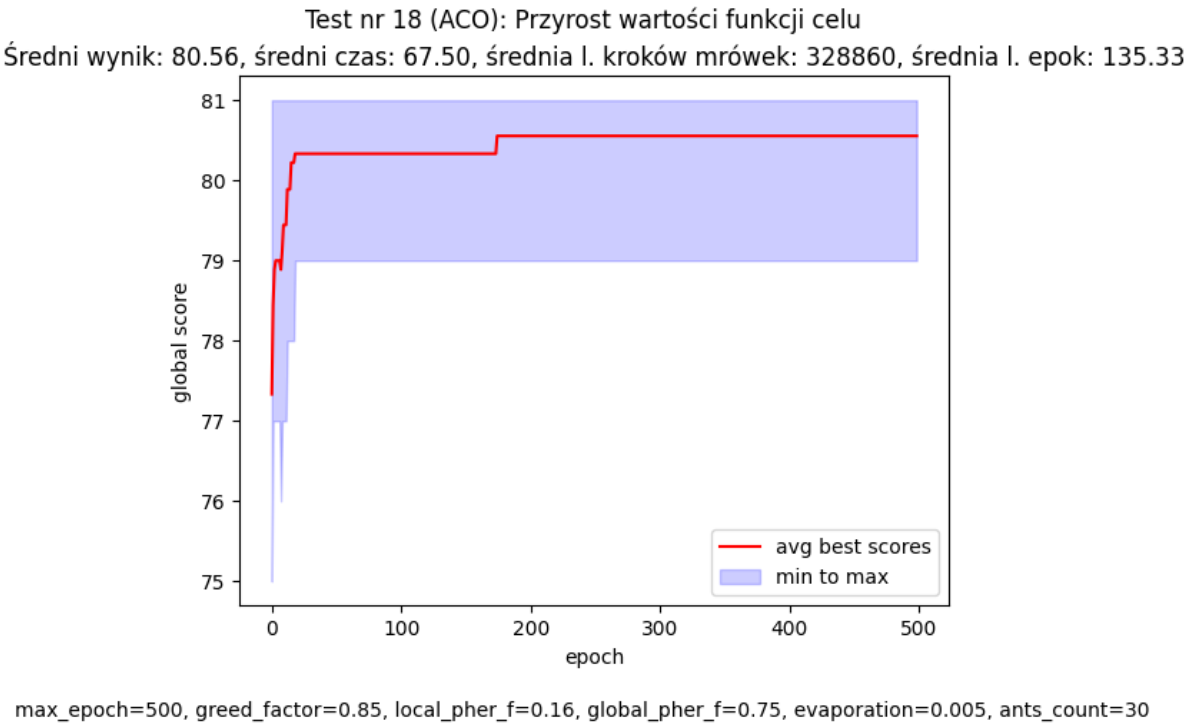
Zwiększenie liczby mrówek do wartości maksymalnej (81) sprawiło, że algorytm rozwiązał każde z zadań w pierwszej iteracji (rys. 4.12). Brzmi to doskonale, jednak wartość średniej liczby kroków mrówek jest tylko nieznacznie mniejsza od wartości z testu nr. 13 (rys. 4.11). Pokazuje to istotność przywiązywania większej uwagi do tego wskaźnika, niż do liczby epok, których złożoność zależy od wielkości "kolonii".



Rys. 4.12. Test nr 15 algorytmu ACO

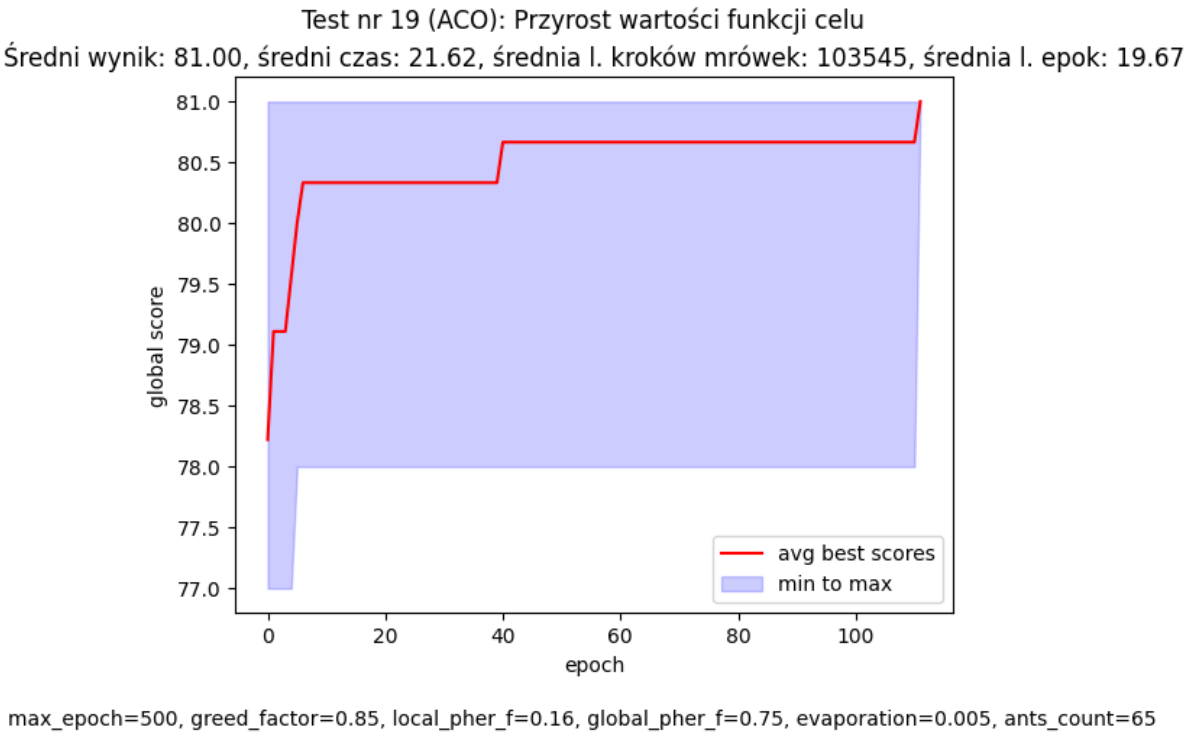
4.2.3. Sudoku - poziom trudny

Działanie algorytmu ACO przetestowane na planszach sudoku sklasyfikowanych jako trudne. Próba przedstawiona na wykresie 4.13 dała zadowalające wyniki, biorąc pod uwagę trudność wejściowych tablic sudoku, w których znajduje się zdecydowanie więcej zer niż było to w przypadku wcześniej omawianych poziomów trudności. Średni wynik znalezionego rozwiązania wyniósł 80,56, co mówi, że niemalże w każdej próbie algorytm poprawnie rozwiązał sudoku. Osiągnięte zostało to jednak kosztem stosunkowo dużego czasu oraz znaczącej liczby epok potrzebnych na skuteczne przeszukanie przestrzeni dostępnych rozwiązań. Dodatkowym utrudnieniem oprócz poziomu zaawansowania sudoku było ograniczenie liczby mrówek "pracujących" nad rozwiązaniem problemu. Oczekiwane jest lepsze działanie algorytmu dla większej kolonii mrówek, co zostało zaprezentowane w kolejnych testach.



Rys. 4.13. Test nr 18 algorytmu ACO

Test nr 19 (rys. 4.14) to kolejny przykład stuprocentowej skuteczności algorytmu mrówkowego. Tym razem, dla nieznacznie zmienionych wartości współczynników feromonów i kolonii 65 mrówek program potrzebował średnio 20 sekund (100000 operacji na tablicy sudoku) do znalezienia rozwiązania. Biorąc pod uwagę wysoką złożoność problemu, wynik ten jest bardzo dobry. Zgodnie z oczekiwaniami, zwiększenie liczby mrówek poprawiło działanie ACO.



Rys. 4.14. Test nr 19 algorytmu ACO

5. Wnioski

Eksperymenty opisane w 2.2.3 pozwoliły na porównanie algorytmów. Wynika z nich, że do rozwiązywania sudoku znacznie lepiej przystosowany jest algorytm ACO (Ant Colony Optimization). Wyniki na wszystkich płaszczyznach pokazały jednoznacznie wyższość tej implementacji nad wynikami algorytmu genetycznego. Dodatkową zaletą metody mrówkowej jest mała liczba hiperparametrów, oraz fakt, że ich zmiana nie wpływa drastycznie na jakość wyników. Początkowe próby algorytmu genetycznego nie dawały obiecujących efektów, zdarzały się nierozwiązywalne plansze - a przynajmniej tak mogłoby się wydawać. Zwiększenie maksymalnej liczby epok i znalezienie optymalnych ustawień sprawiło jednak, że wyniki **skuteczności** były znacznie lepsze. Kosztowało to niestety czas - podczas poszukiwań najlepszych nastaw parametrów rozwiązanie 25 plansz potrafiło trwać niemal godzinę. Jedną z zauważonych cech algorytmu genetycznego była jego powtarzalność. Jest to niewątpliwie pozytywna, oraz często pożądana cecha w zadaniach przeszukiwania przestrzeni. Po nabyciu empirycznej wiedzy i umiejętności rozwiązywania danego typu problemów przy wykorzystaniu algorytmu genetycznego, jesteśmy w stanie określić spodziewany czas i/lub maksymalną liczbę iteracji reprodukcji. W algorytmie Ant Colony Optimization, co widoczne jest na niektórych wykresach, wygenerowano stany początkowe prowadzące do rozwiązania już w pierwszej epoce. Taki wynik jest doskonały. Jednakże wykryto też przypadki, w których po 400 epokach zadanie wciąż nie było rozwiązane. Oba przpypadki potrafiły pojawić się dla tych samych wartości hiperparametrów, co świadczy o bardzo wysokiej niedeterministyczności algorytmu. Przy wykorzystaniu niepełnej puli "mrówek" (czyli przy dobraniu liczby mrówek mniejszej od liczby wszystkich pól na planszy) wiele

zależy od wylosowanych pól startowych. Algorytm ewolucyjny również jest bardzo zależny od populacji początkowej, co jednak zostało zniwelowane doborem odpowiednich hiperparametrów i wybraniem skutecznych metod krzyżowania i mutacji. Testy wykazały słuszność implementacji parametru *succession_rate*. Można zauważyć pozytywny wpływ dodania takiego niedeterministycznego elementu na uciekanie z optimów lokalnych. Jest to pozytywne zjawisko, ponieważ to właśnie liczne i ciężko rozpoznawalne optima lokalne są największą trudnością sudoku. Algorytm dążący do maksymalizacji funkcji celu potrafi na dobre utknąć w jednym z takich optimów, uzyskując bardzo wysoki wynik funkcji, podczas gdy w rzeczywistości jest bardzo daleko od wypełnienia "wygrywającego". W algorytmie genetycznym zastosowano resetowanie populacji przy braku progresu w k epokach z rzędu, a w ACO zastosowano ewaporację wartości w tablicy feromonów.
Działanie algorytmów pokazuje wyższą tendencję algorytmu genetycznego do wpadania w optima lokalne. Jego zaletą jest natomiast szybkie uciekanie z nich (słaba lokalizacja optimów, dobra eksploracja przestrzeni poszukiwań). Algorytm Ant Colony Optimization rzadziej ląduje w optimach lokalnych, ale też ma znacznie większe problemy z uciekaniem z nich - mała kowariancja wyników.

6. Struktura projektu i reprodukcja testów

Stworzony projekt został podzielony według następującej struktury (kody używane w projekcie zawarte są w katalogu *src* projektu):

— katalog *problem*:
  — katalog *sudoku_boards*, w którym znajdują się wygenerowane wcześniej tablice sudoku, służące do testów, w postaci tablic *numpy* zapisanych w formacie .npy,
  — plik *sudoku_manager.py*, zawiera implementację generatora tablic sudoku (tu wykorzystanie biblioteki *dokusan*) oraz klasy Sudoku, wczytującej tablicę z podanego pliku, a także posiadającej metody oraz atrybuty wykorzystywane w tworzonych algorytmach,
— katalog *solvers*:
  — katalog *ga* – zawiera implementację solvera algorytmu genetycznego – plik *ga_solver.py*,
  — katalog *aco* – zawiera implementację solvera algorytmu ACO oraz klasy Ant (czyli instancji mrówki przeszukującej planszę sudoku) – pliki *aco_solver.py* i *ant.py*,
— katalog *tests* – znajdują się w nim pliki *tests_ga.ipynb* i *tests_aco.ipynb*, które są notatnikami *Jupyterowymi*, służącymi do łatwiejszego i sprawniejszego testowania stworzonych rozwiązań; skrypty w nich stworzone uruchamiają działanie danego algorytmu dla określonych parametrów początkowych (w tym wybranego ziarna losowości), które są zapisywane (tak jak i generowane wykresy) w katalogu powyżej *src* nazwanym *results*,
— katalog *reproduce* – zawiera skrypty do odtworzenia wybranej próby testowej – pliki *reproduce_ga.py* i *reproduce_aco.py*.

W celu odtworzenia wybranej próby testowej, należy przejść do katalogu *reproduce* i wprowadzić nazwę jednego z istniejących plików, z zapisanymi parametrami testu, jako zmienną **TEST_FILE**. Pliki z zapisanymi parametrami poszczególnych testów znajdują się w katalogu powyżej *src* nazwanym *results*, a dalej w folderze *params* umieszczone są katalogi *ga* oraz *aco* z odpowiednio nazwanymi plikami rozszerzenia .json, które przechowują słowniki z nastawami testów. Po uruchomieniu pliku dla wybranej próby, odtworzony zostanie test, a wyniki zostaną zaprezentowane w postaci wykresu, na którym wyświetlone będą wszystkie, kluczowe z perspektywy badanego algorytmu, wartości wyjściowe.

Dokładniejsza instrukcja zbudowania i uruchomienia projektu znajduje się w pliku **README.md** załączonym do repozytorium projektowego.