Marcel Buczkowski 145159

Zaawansowane programowanie
Sprawozdanie

Tworzenie aplikacji okienkowej w języku C# oraz implementacja metaheurystycznego algorytmu genetycznego w celu rozwiązywania problemu mapowania poprzez hybrydyzację *m* fragmentów DNA do *n* próbek.

Poznań, 19.01.2023

Problem IV – MIN ERROR MAP

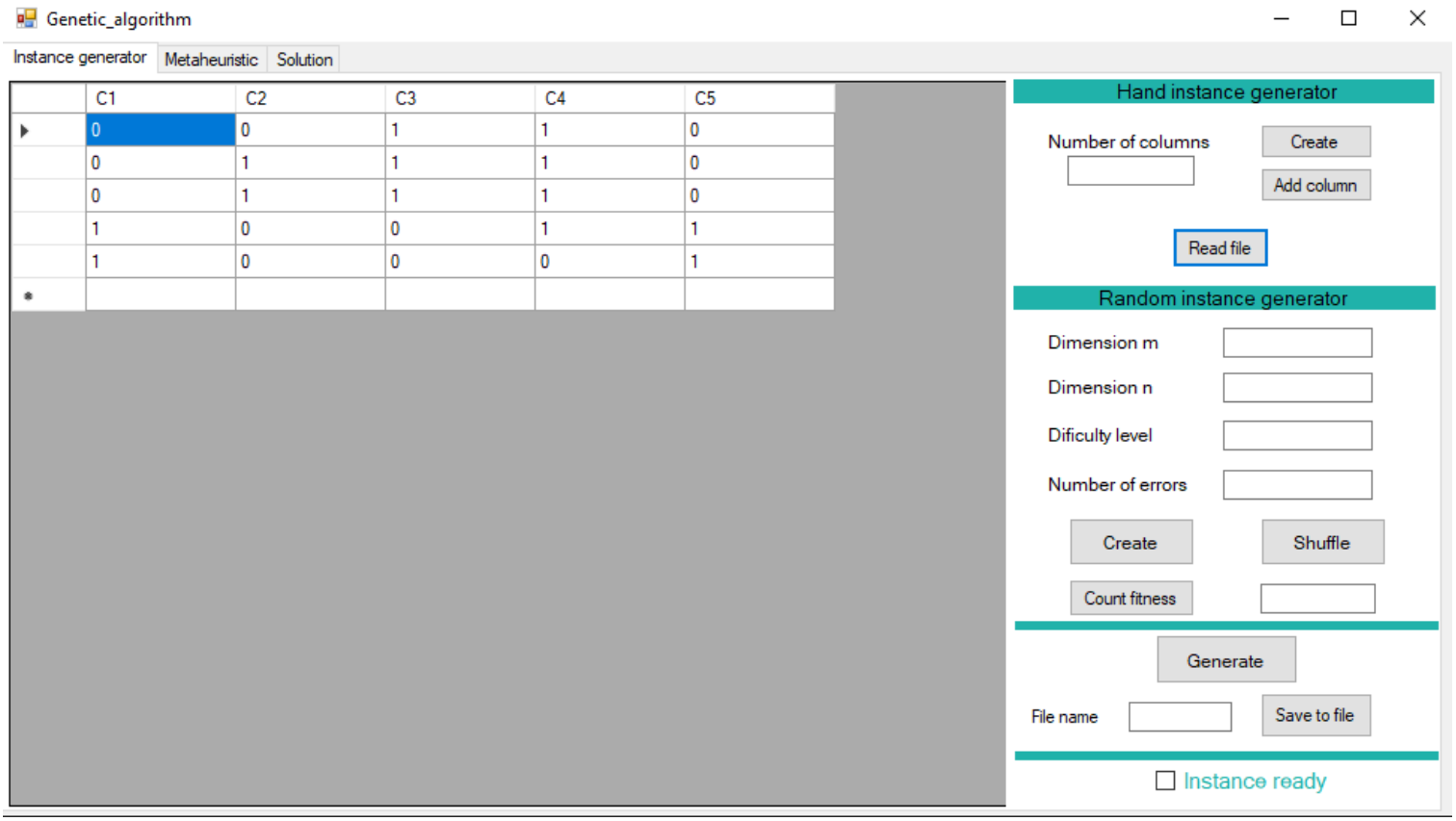
Instancja: Binarna macierz M_{mxn} hybrydyzacji m fragmentów do n próbek.

Rozwiązanie: Uporządkowanie kolumn macierzy M takie, że liczba pól, których wartość należy zmienić, aby macierz osiągnęła własność consecutive 1s w wierszach jest minimalna.

1. Opis aplikacji

Aplikacja zbudowana jest z jednego okienka podzielonego na 3 różne zakładki, którymi są generator instancji, metaheurystyka oraz rozwiązanie. W celu rozwiązania problemu aplikacja wykorzystuje wielowątkowość i działa na dwóch niezależnych wątkach. Zaimplementowana w aplikacji metaheurystyka to algorytm genetyczny.

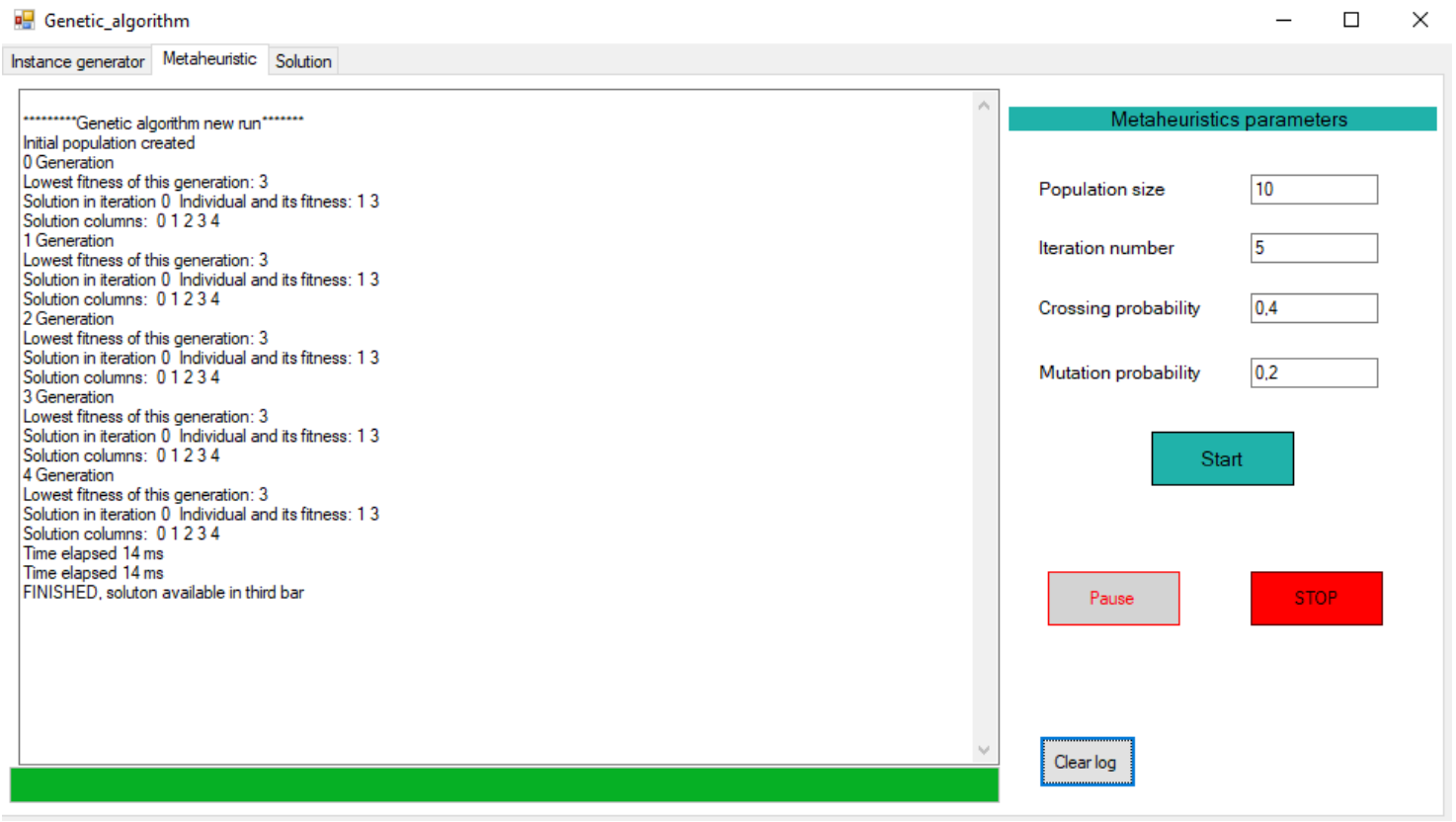
1.1. Generator instancji



Okno podzielone jest za pomocą *SplitContainer* na dwie części. Po prawej stronie znajduje się panel główny, który służy do tworzenia nowych instancji lub wczytywania wcześniej stworzonych. Po lewej natomiast wizualizowana jest dana instancja za pomocą *DataGridView.*

Panel główny pozwala na dwa sposoby generowania instancji – ręczne oraz zautymatyzowane. Aby wygenerować instancję ręcznie należy w wyznaczonym miejscu wpisać liczbę kolumn oraz wcisnąć przycisk *Create.* Wówczas tworzony jest pusty obiekt *DataTable* wiązany do *DataGridView.* Tabela ma zadaną liczbę kolumn, natomiast liczba wierszy automatycznie zwiększa się wraz z ich zapełnianiem. Dodatkowo można użyć przycisku *Add column*, aby dodać kolejną kolumnę do tabeli.
Drugim sposobem jest automatyczne tworzenie instancji o zadanych parametrach, które należy wpisać ręcznie w wyznaczone miejsca. Parametry to wymiary *m* i *n* macierzy, dodatkowe kryterium trudności czyli maksymalna możliwa ilość 1 w wierszu, oraz ilość wprowadzanych błędów. Po naciśnięciu przycisku *Create* sprawdzana jest poprawność parametrów, oraz to czy w ogóle zostały wprowadzone. Jeśli nie to wyświetlane są stosowne komunikaty, jeśli tak to tworzony jest obiekt jednej z dwóch głównych klas programu – klasy *Instance* (opisanej w późniejszej części sprawozdania).
Po stworzeniu obiektu z danymi parametrami macierz jest w odpowiedni sposób wypełniana, a następnie konwertowana do *DataTable*, aby ją zwizualizować.
Przycisk *Count fitness* pozwala na obliczenie wartości funkcji celu stworzonej macierzy, aby wiedzieć dokładnie jaką wartość można uznać za rozwiązanie w danej instancji (żeby upewnić się czy założenie 1 błąd = -1 wartości funkcji celu). W dodatku pozwala sprawdzić fitness macierzy, która jest przekazana metaheurystyce. Przycisk *Shuffle* pozwala na losową zmianę kolejności kolumn w macierzy. Jest to kluczowe, ponieważ uprzednio wygenerowana macierz jest rozwiązaniem instancji, przemieszanie jej kolumn tworzy rzeczywistą instancję – macierz, która będzie przekazana na wejście algorytmu metaheurystycznemu.
Jeszcze inną opcją jest odczytanie instancji z pliku. Po wciśnięciu przycisku *Read file* otwierane jest okno dialogowe, z którego należy wybrać plik do odczytu. Następnie macierz z pliku konwertowana jest to *DataTable,* a ta wiązana z *DataGridView.*
Gdy już instancja została wygenerowana lub odczytana, kluczowym przyciskiem jest *Generate*. Po jego wciśnięciu (jeśli tabela danych istnieje i jest większych wymiarów niż 2x2) wypełniona *DataTable* konwertowana jest do globalnej macierzy *final_matrix,* na której później pracować będzie algorytm genetyczny. Dodatkowo tworzony jest globalny słownik *final_columns*, aby zapamiętać każdą kolumnę i jej indeks, co pozwoli później na odtworzenie macierzy będącej rozwiązaniem (rozwiązanie to sekwencja liczb oznaczających kolejność kolumn).
Dzięki przyciskowi *Save to file* możliwa jest również opcja zapisu wygenerowanej instancji do pliku o zadanej nazwie.
Na samym dole znajduje się *check box,* który trzeba zaznaczyć w momencie kiedy uzna się że instancja jest gotowa i wygenerowana (za pomocą przycisku *Generate*). Bez tego metaheurystyka nie zacznie obliczeń, a wprowadzanie zmian w instancji automatycznie odznacza *check box.*

1.2. Metaheurystyka

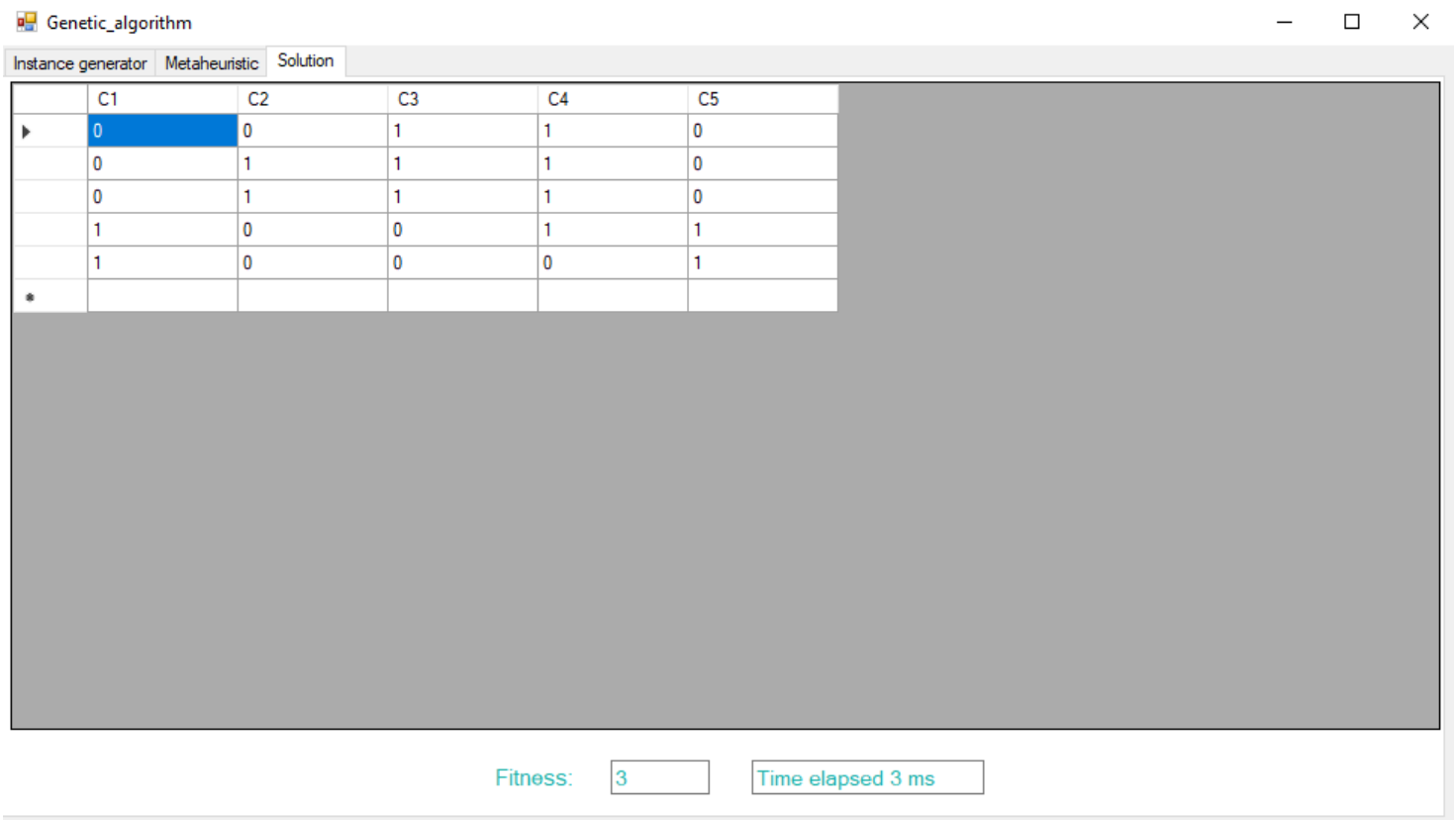


Kolejna zakładka to interfejs metaheurystyki. Również podzielona jest na dwie części. Po prawej stronie znajduję się panel służący do podawania parametrów dla metaheurystyki, oraz przyciski do obsługi jej działania. Po lewej natomiast obecny jest *rejestr zdarzeń (log)* informujący o działaniu metaheurystyki i jej postępach. Pod spodem znajduję się także *progres bar*, który zapełnia się kolorem zielonym w miarę działania metaheurystyki i również informuje o jej postępach.

Po wciśnięciu przycisku *Start* sprawdzane jest czy wprowadzono parametry metaheurystyki, a także ich poprawność jeśli tak. Jeśli wprowadzono poprawne wartości (nieujemne lub nie większe niż 500 (rozmiar populacji), 300 (liczba iteracji) i 1 dla pozostałych parametrów) to metaheurystyka zostaje uruchomiona. Zanim to jednak nastąpi inicjowana jest liczba maksymalna dla *progress bara* – liczba iteracji oracz początkowa i minimalna liczba obie równe 0. W tym miejscu również tworzony jest obiekt *BackgroundWorker,* który umożliwia działanie metaheurystyki na innym (pobocznym) wątku, niż wątek główny programu. Dzięki temu w trakcie jej obliczeń, użytkownik aplikacji może z niej swobodnie korzystać np. generując instancję. Do *BackgroundWorkera* za pomocą delegata wysyłane są procedury, które ma wykonać. Także w tym miejscu rozpoczyna się liczenie czasu działania metaheurystyki oraz jej inicjowanie. Opiera się ona na drugiej głównej klasie programu – klasie *Genetic Algorithm* (dokładnie opisanej w późniejszej części sprawozdania). Najpierw tworzony jest obiekt poprzez podanie macierzy *final_matrix* oraz parametrów metaheurystyki do

konstruktora klasy. Następnie algorytm działa zadaną liczbę iteracji, i w każdej wywołuje odpowiednie metody na obiekcie klasy *Genetic Algorithm* a także wysyła informacje o postępach za pomocą *BackgroundWorker.ReportProgress* - dzięki niemu możliwe jest przesyłanie danych z wątku pobocznego do wątku głównego, w którym utworzono *Log* oraz *progress bar* i wówczas odpowiednie informacje wyświetlane są w *rejestrze zdarzeń* a *progres bar* się wypełnia.
W każdej iteracji wyświetlany jest najniższy fitness danej generacji, a także numer i dostosowanie osobnika będącego rozwiązaniem oraz samo rozwiązanie jako sekwencja kolumn wraz z informacją z której iteracji to rozwiązanie pochodzi.
Po zakończeniu obliczeń w log'u wyświetlony zostaje czas tych obliczeń oraz informacja, że rozwiązanie widoczne jest w trzeciej zakładce.
Istotny jest także przycisk *Pause,* który po wciśnięciu zmienia wartość globalnej zmiennej *volatile bool Active* (domyślnie ustawionej na true) na wartość *false,* co wstrzymuje obliczenia metaheurystyki na wątku pobocznym. Dodatkowo sam przycisk zmienia nazwę na *Resume,* a po jego ponownym wciśnięciu obliczenia są wznawiane.
Natomiast po wciśnięciu przycisku *Stop* wartość globalnej zmiennej *volatile bool Stop* (domyślnie ustawionej na *false*) zmienia się na wartość *true,* co powoduje przerwanie pętli głównej odpowiedzialnej za obliczenia na wątku pobocznym i tym samym zakończenie tych obliczeń ze zwróceniem wyniku uzyskanego do momentu ich zakończenia.
Przycisk *Clear Log* wymazuje całą zawartość *rejestru zdarzeń*.

1.3. Rozwiązanie



Trzecią zakładkę stanowi *DataGridView* na którym zobrazowane jest rozwiązanie obliczone przez algorytm genetyczny. *DataTable* z rozwiązaniem tworzona jest jeszcze w trakcie trwania wątku pobocznego, po zakończeniu obliczeń. Kolejność kolumn otrzymana przez metaheurystykę jest zamieniana do macierzy wynikowej poprzez zastosowanie wcześniej wspomnianego globalnego słownika *final_columns*. Dodatkowo poniżej zamieszczona jest wartość funkcji celu rozwiązania oraz czas wykorzystany do jego wyliczenia.

2. Opis algorytmu

2.1. Generator instancji

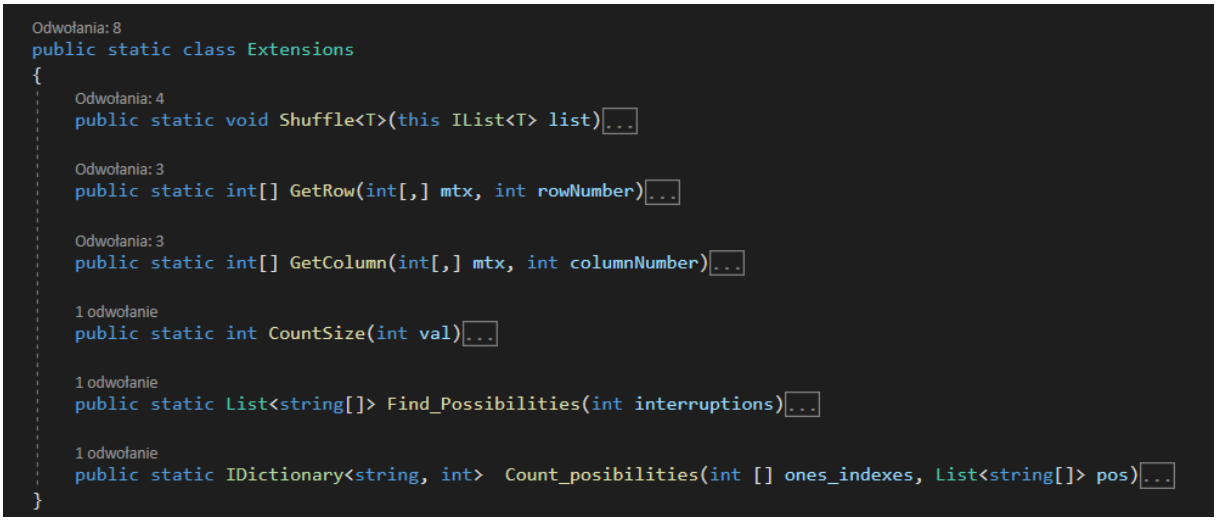
```
Odwołania: 3
class Instance
{
    private int dimension_m;
    private int dimension_n;
    private int error_number;
    private int density;
    private int[,] matrix;

    1 odwołanie
    public Instance(int dim_m, int dim_n, int err, int dens)...

    Odwołania: 2
    public void FillMatrix()...

    1 odwołanie
    public void Add_errors()...

    1 odwołanie
    public void Check_columns()...

    1 odwołanie
    public void ShuffleColumns()...

    1 odwołanie
    public int ComputeSolutionFitness()...

    Odwołania: 2
    public void ConvertToDtable()...
}
```

Generator instancji opiera się o klasę *Instance,* która zawiera wszystkie niezbędne metody i atrybuty do prawidłowego generowania instancji. Tworząc obiekt tej klasy za pomocą przycisku *Generate* konstruktor inicjuje zmienne oraz tworzy własną macierz na wzór przekazanej mu jako parametr wejściowy. Tworzone zmienne to wymiary macierzy – *dimension_m* oraz *dimension_n*, *error_number* – liczba wprowadzonych błędów oraz *density* – czyli możliwa maksymalna gęstość zapełnienia wiersza jedynkami.
Kolejno wywołana zostaje metoda *Fill matrix:*
Funkcja dla każdego wiersza losuje możliwą ilość jedynek od *density - 2* do *density.* Następnie wylicza maksymalny indeks, od którego może wklejać zadaną liczbę jedynek i sprawdza czy indeks iterowanego wiersza jest mniejszy od tego indeksu. Jeśli tak to dokonuje jednego z dwóch równie prawdopodobnych wyborów. Albo wkleja jedynki od indeksu równemu indeksowi wiersza, albo wkleja je od początku wiersza. Jeśli jednak indeks aktualnego wiersza jest większy od maksymalnego indeksu wklejania jedynek, to również dokonuje jednego z dwóch wyborów. Albo wkleja jedynki od tego maksymalnego indeksu do końca wiersza, albo od początku wiersza. W ten sposób otrzymywana jest macierz bez błędów. Kolejno dodawane są błędy do instancji.
Metoda *Add errors:*
Najpierw wyliczana jest liczba błędów negatywnych i pozytywnych, i ich ilość jest równa, bądź błędów negatywnych będzie więcej o 1 (w zależności czy liczba błędów ogólnie jest parzysta czy nie).

Kolejno tworzona jest tablica wartościami od 0 do *dimension_m*, która zostaje przemieszana – i w ten sposób reprezentuje losową kolejność dodawania błędów w wierszach. Następnie dla każdego wiersza dodawane są dane ilości błędów pozytywnych z zachowaniem jakościowego ich wprowadzania, to znaczy nie wprowadza się błędów na granicy ciągów zer i jedynek. Osiągane jest to przez zainicjowanie dodatkowych tablic – informujących o indeksach zer i jedynek w danym wierszu. Jeśli długość tablicy z indeksami zer jest równa jeden to nie da się wprowadzić błędu pozytywnego w tym wierszu gdyż jedynki zapełniłyby wówczas cały wiersz. Kolejno dla każdego indeksu zera w wierszu sprawdzane jest czy indeks ma wartość 0 (lewa skrajna pozycja) i wówczas czy na sąsiadującej pozycji jest zero, jeśli tak można dodać błąd. Jeśli nie to sprawdzane jest czy indeks ma wartość maksymalna (prawa skrajna pozycja) i wówczas czy wartość go poprzedzająca jest równa 0. Jeśli tak to można dodać błąd, jeśli nie to sprawdzana jest ostatnia opcja, czyli dla indeksu będącego innym niż pierwszy i ostatni sprawdzane jest czy wartości sąsiadujące są równe 0. Jeśli tak błąd zostaje dodany. Kolejno tablica z kolejnością wierszy jest przemieszana, aby dodane błędy negatywne nie znalazły się koniecznie w tych samych wierszach. Dodawanie błędów negatywnych przebiega w analogiczny sposób, z tym że błędu negatywnego nie można dodać jeśli liczba jedynek w wierszu jest równa 2.
Jeśli w skrajnie małych i prostych instancjach nie da się wprowadzić zadanej liczby błędów to zwracany jest wyjątek.
Następnie sprawdzana jest poprawność kolumn przez metodę *Check columns:*
Dla każdej kolumny sprawdza czy przypadkiem któraś z nich nie jest wypełniona samymi zerami (co jest niepożądane). Jeśli tak, to macierz wypełniana jest na nowo i dodawane są od nowa błędy.
Wszystko co generator zrobił do tej pory to wygenerowanie instancji z błędami, która będzie docelowym rozwiązaniem, a jej wartość funkcji celu będzie wartością docelową, do której dążyć będzie metaheurystyka. Zanim jednak macierz będzie przekazana metaheurystyce, należy w losowy sposób przemieszać jej kolumny. W ten sposób wygenerowana instancja nie przekazuje żadnej informacji dotyczącej optymalnego (z założenia) rozwiązania metaheurystyce, a możliwe jest stwierdzenia jakości rozwiązań przez nią zwracanych (porównując wartość funkcji celu z naszym założonym rozwiązaniem optymalnym).
Metoda *Shuffle columns:*
Aby zmienić kolejność kolumn w macierzy najpierw tworzona jest dwuwymiarowa tablica dodatkowa, która przechowuje transpozycję tej macierzy. Kolejno generowana jest tablica z wartościami od 0 do *dimension_n* i zostaje przemieszana. W ten sposób otrzymany zostaje nowy losowy porządek kolumn w macierzy wyjściowej równoznaczny z nowym porządkiem wierszy w macierzy transpozowanej. Macierz wyjściowa odtwarzana jest poprzez transpozycje macierzy transpozowanej z zachowaniem nowego porządku.

W ten sposób otrzymana macierz gotowa jest do przekazania metaheurystyce.
Kolejną metodą jest metoda *Compute Solution Fitnes,* która w zasadzie skopiowana została z metody metaheurystyki, tylko po to, aby móc sprawdzić dostosowanie wygenerowanej automatycznie macierzy przed i po zmianie kolejności kolumn w macierzy.
Ostatnią metodą tej klasy jest *Convert to DTable,* odpowiedzialna za konwersję macierzy do DataTable, która może być zwizualizowana w oknie programu za pomocą DataGridView.

2.2. Metody pomocnicze



W klasie *Extensions* umieszczono rożne metody pomocnicze, do których często odwołuje się program w trakcie obliczeń i które są odpowiedzialne za istotne dla działania programu operacje.
Metoda *Shuffle* otrzymuje listę oraz miesza kolejność jej elementów w sposób losowy.
Metody *GetRow* i *GetColumn* przyjmują macierz i zwracają odpowiednio jej wiersz lub kolumnę w postaci tablicy.
Kolejne metody używane są przez metaheurystykę w celu obliczania funkcji celu i są w tym procesie kluczowe, dlatego ich działanie zostanie opisane przy charakterystyce algorytmu metaheurystycznego w dalszej części sprawozdania.

2.3. Algorytm genetyczny

```
class GeneticAlgorithm
{
    private int[,] matrix;
    private int dimension_m;   //rows
    private int dimension_n;   //cols
    private List<int[]> population;
    private List<int> fitness;
    private List<int[]> mutations;  //nr osobnika, col, col
    private IDictionary<int, int[]> columns;
    private int population_size;
    private double crossing_probability;
    private double mutation_probability;
    private int[] solution;
    private int[] solution_sequence;

    1 odwołanie
    public GeneticAlgorithm(int[,] instance, int pop_size, double cros, double mut)...

    1 odwołanie
    public void Create_starting_population()...

    1 odwołanie
    public void Compute_fitness()...

    1 odwołanie
    public string print_fitness()...

    1 odwołanie
    public string print_solution()...

    1 odwołanie
    public string print_solution_sequence()...

    Odwołania: 2
    public int give_solution_fitness()...

    1 odwołanie
    public List<int> Selection()...

    1 odwołanie
    public void Reproduction() ...

    1 odwołanie
    public void Mutation()...

    1 odwołanie
    public int[] return_solution_sequence()...
}
```

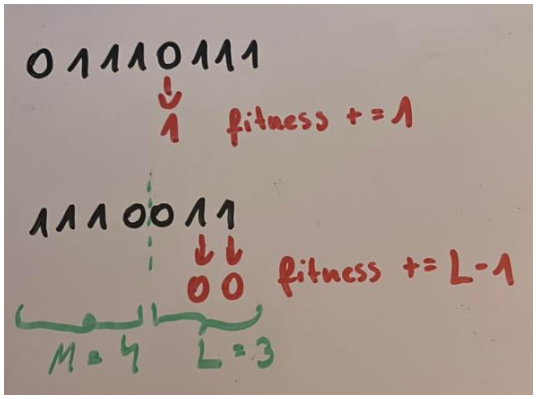
Metaheurystyka oparta jest o klasę *Genetic Algorithm* zawierającą szereg metod i atrybutów. Metody główne to: *Create starting population, compute fitness, selection, reproduction oraz mutation.* Reszta metod służy do zwracania wartości np. Fitnessu czy tez rozwiązania do obsługi rejestru zdarzeń oraz wizualizacji rozwiązania. Klasa korzysta także z metod zewnętrznych z klasy *Extensions.*

Po stworzeniu obiektu tej klasy za pomocą przycisku *Start* generator instancji otrzymuje podane parametry metaheurystyki oraz macierz na której algorytm będzie bazować. Dodatkowo tworzy słownik *Columns,* który przechowuje kolumny macierzy wraz z ich numerami.

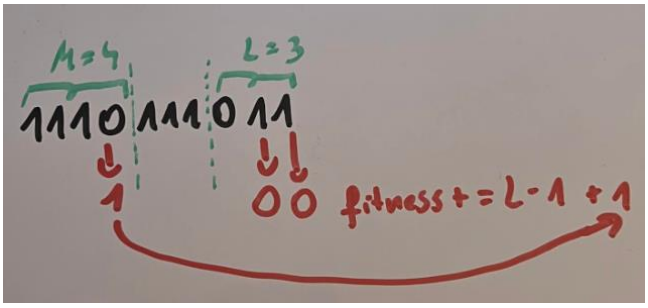
Pierwsza wywołana metoda to *Create starting population:*

Tworzy ona tablicę liczb od 0 do *dimension_n* (liczba kolumn). Kolejno w pętli od 0 do *population_size* tablica ta jest mieszana, a jej kopia dodawana do listy tablic *population.* W ten sposób otrzymana jest populacja, w której każdy osobnik posiada przemieszaną tablicę reprezentującą kolejność kolumn (możliwe są identyczne osobniki, choć to raczej możliwe tylko dla małych instancji, z których tworzone są duże populacje).

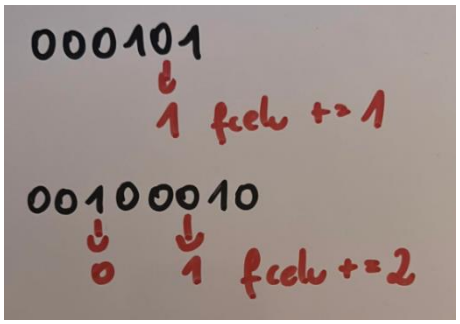
Kolejna wywołana metoda to *Compute fitness:*
Jest to największa i najbardziej skomplikowana metoda w programie. Działa w głównej pętli od 0 do rozmiaru populacji, czyli dla każdego osobnika odtwarza jego macierz za pomocą jego tablicy z kolejnością kolumn, oraz słownika przechowującego kolumny. Następnie iteruje po każdym wierszu macierzy obliczając jego funkcję celu. Niezbędne jest do tego wydobycie indeksów zer i jedynek w każdym wierszu do dwóch osobnych tablic *ones_indexes* oraz *zeros_indexes.* Kolejno obliczana jest liczba przerwań ciągów jedynek w wierszu. Następnie jeśli jest większa od 0, musi sprawdzić się któraś z poniższych opcji:
a) Liczba jedynek w wierszu równa jest dimension_n – 2:
Jeśli tak to sprawdzana jest ilość przerwań. Jeśli jest jedno i zero występuje na początku lub na końcu wiersza to fitness += 1 (drugie zero musi być gdzieś indziej w wierszu). Jeśli natomiast przerwanie jest jedno i występuje gdzieś w środku wiersza to znajdowany jest indeks pierwszego 0 i wiersz dzielony jest na 2 tablice za tym pierwszym 0. Ftiness równy jest wówczas długości krótszej tablicy - 1.



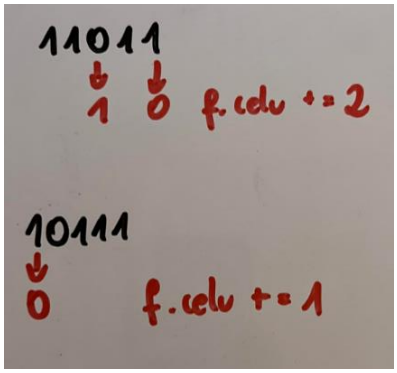
Jeśli liczba przerwań jest większa od 1 (czyli maks. 2) to wynajdywane są 2 podtablice. Jedna do pierwszego 0 (włączając je), a druga od drugiego zera (również je włączając). Fitness zwiększany jest wówczas o długość krótszej podtablicy.



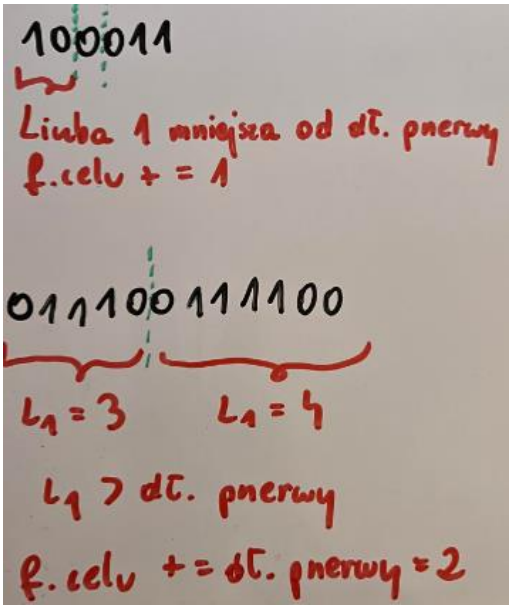
b) Liczba jedynek w wierszu równa jest 2:
Jeśli te jedynki rozdziela jedno zero to f. celu += 1, natomiast jeśli więcej to f. celu zawsze += 2



c) W wierszu występuje jedno przerwanie:
(Jeśli warunek a) jest spełniony to reszta nie jest sprawdzana itd.)
Jeśli w przerwaniu jest tylko jedno 0, i znajduje się na drugim lub przedostatnim miejscu, to f. celu += 1. Jeśli jest gdziekolwiek indziej to f. celu += 2.



Jeśli w jednym przerwaniu jest więcej zer, to wynajdywane są indeksy rozpoczęcia oraz zakończenia przerwy i jej długość. Na podstawie tych wartości wyliczany jest *splitter_ind,* czyli indeks przecięcia przerwy w jej połowie. Po tym indeksie tworzone są dwie podtablice. Następnie sprawdzane jest czy liczba jedynek w podtablicy gdzie jest ich mniej jest mniejsza od liczby zer będących przerwaniem. Jeśli tak, to f. celu += liczba tych jedynek. Jeśli nie, to f. celu += liczba zer w przerwaniu (plus 1 jeśli zamiana tych zer w jedynki oznacza, że cały wiersz wypełniony jest jedynkami).



d)W wierszu występuje więcej niż jedno przerwanie:
Sprawdzane jest czy są to dwa przerwania czy więcej. Jeśli dwa to ręcznie tworzona jest lista *posibilities*, która sprawdza wszystkie możliwe kombinacje 'naprawy' danego wiersza w celu doprowadzenia do własności *consecutive 1s (brak przerwań w bloku jedynek w wierszu).* Równocześnie zliczany jest koszt każdej kombinacji, a do funkcji celu dodawany jest ten najniższy. Kombinacje są zakodowane w poniższy sposób:

- 1, 21 – oznacza zamianę pierwszej przerwy na jedynki, oraz jedynek za drugą przerwą na zera
- 1, 2 – oznacza zamianę obu przerw na jedynki
- 10, 2 – oznacza zamianę jedynek przed pierwszą przerwą na zera, oraz drugiej przerwy na jedynki
- 10, 21 – oznacza zamianę jedynek przed pierwszą przerwą na zera, oraz jedynki za drugą przerwą na zera
- 11, 21 – oznacza zamianę jedynek za pierwszą przerwą na zera, oraz jedynek za drugą przerwą na 0

Zatem pojedyńcza cyfra oznacza przerwę, natomiast w liczbach dwucyfrowych pierwsza z nich oznacza przerwę a druga to zawsze 0 i 1 odpowiednio oznaczające przed i za.
Jeśli przerwań jest więcej niż dwa, to w celu stworzenia listy *posibilities* używane są tu dwie funkcje wcześniej wspomniane w klasie *Extensions* metod dodatkowych.
Pierwsza z nich to *count_size,* która przyjmuje liczbę przerwań i zwraca rozmiar listy *posibilities.* Rozmiar ten zależny jest od liczby przerwań w taki sposób, że dla 3 przerwań lista ma rozmiar 6, dla 4 rozmiar 10, dla 5 rozmiar 25 i dla 6 rozmiar 21 itd. Następnie w rozbudowanej zapętlonej procedurze find_posibilities dodawane są kombinacje. Dla liczby przerwań 3 funkcja tworzy takie kombinacje jak: 1,2,3 1,2,31 1,21,31 10,2,3 10,2,31 10,20,3.
Dla 4 przerwań następujące kombinacje: 1,2,3,4 1,2,3,41 1,2,31,41 1,21,31,41 10,2,3,4 10,2,3,41 10,2,31,41 10,20,3,4 10,20,3,41 10,20,30,4.
I zwiększając liczbę przerwań wzorzec jest naśladowany.
Jeśli liczba przerwań w wierszu jest większa od 9, to kodowanie wykonywane jest w nieco inny sposób. Wówczas 10 oznacza dziesiąte przerwanie, a -10 oznacza ciąg jedynek przed pierwszym przerwaniem. Pojawiają się wówczas również 3 cyfrowe kombinacje. Jednak sposób ich interpretacji i wyliczanie fitnessu dla każdej kombinacji realizowane jest przez kolejną funkcję *Count posibilities*, która zwraca słownik z każdą liczbą danej kombinacji i jej kosztem. Do funkcji przekazywana jest lista z numerami indeksów jedynek w wierszu i na jej podstawie wyliczane są dodatkowe listy – długości przerwań oraz indeksów przerwań. Za pomocą tych 3 obiektów wyliczana jest wartość funkcji celu dla każdej wartości pochodzącej z danej kombinacji. Finalnie np. powstaje słownik 1: 3, 2: 1, 3: 5, 21: 2 itd. gdzie kluczem jest liczba (numer przerwania i ew. oznaczenie za/przed) z kombinacji a wartością jest zmiana fitnessu. Procedura liczenia wywoływana jest dla każdego wiersza, i wybierana jest każdorazowo najmniejsza wartość. Sumując wartości dla każdego wiersza otrzymywany jest fitness osobnika, który zostaje zapisany na liście *fitness.* Zapamiętywane zostaje również rozwiązanie, jeśli fitness danego osobnika jest niższy niż tego uprzednio liczonego.

Kolejna wywoływana funkcja to *Selection:*
Selekcja oparta jest o *metodę ruletki.* Najpierw przeliczane są wartości fitnessu na odpowiednie wartości reprezentujące wielkość pola ruletki, w taki sposób żeby najmniejsza wartość fitnessu miała największe pole. Następnie wartości te zamieniane są na listę prawdopodobieństw wylosowania każdego osobnika, poprzez podzielenie każdego pola ruletki przez sumę wielkości tych pól. Po obliczeniu liczby osobników dopuszczonych do krzyżowania (parametr algorytmu), losowana jest dana ilość metodą prawdopodobieństwa kumulatywnego bez powtórzeń. Wylosowane osobniki zapisywane są na liście *Selected,* która kolejno użyta jest przez metodę *Reproduction*:
Jako, że algorytm ma do czynienia z permutacjami elementów, to reprodukcja odbywa się metodą częściowego odwzorowania. Wybrane osobniki z listy Selected są losowo dobierane w pary, a następnie krzyżowane. Wydobywane są dla każdego osobnika chromosomy, przez co rozumie się sekwencje kolejności jego kolumn. Dla każdej pary losowane są 2 punkty chromosomu (w stałej od siebie odległości równiej 0.4 * dimension_n), które stanowią fragment odwzorowania. Wszystkie elementy w obrębie tego fragmentu tworzą wzór np.
14|32|0
23|14|0
Tworzy odwzorowanie 3 -> 1, 1 -> 3, 2 -> 4, 4 -> 2
Jeśli w odwzorowaniu występują takie cyfry, że np. 1->3, 3->4, to w efekcie powstaje zmiana 1->4. Procedura wykonywana jest w ten sposób, że najpierw do słownika *initial_subs* dodawane są te pierwsze odwzorowania, czyli kluczem jest cyfra z pierwszego chromosomu, a wartością cyfra jej odpowiadająca. Następnie sprawdzane jest czy w stworzonym w ten sposób słowniku jakakolwiek cyfra będąca wartością jest też kluczem. Jeśli tak to odpowiednia procedura przekształca ten słownik w taki sposób, aby odwzorowania były poprawne. Jest to kluczowe, ponieważ niepoprawny zapis odwzorowań będzie skutkował powtarzającymi się cyframi w chromosomie, a co za tym idzie niepoprawnym osobnikiem (rozwiązaniem) w populacji.
Po zamianie obu chromosomów w danej parze rodziców nowe uporządkowania kolumn zapisywane są pod odpowiednimi indeksami na liście głównej *population.*
Ostatnia metoda to *Mutation:*
Jej zadaniem jest najpierw obliczenie liczby osobników, które mają być poddane mutacji (z parametru wejściowego metahurystyki), a następnie wybrać w sposób losowy bez powtórzeń zadaną liczbę osobników z populacji. Mutacja odbywa się w taki sposób, że losowane są dwa różne indeksy od 0 do dimension_n, a następnie w danym osobniku dwie kolumny o wylosowanych indeksach zamieniają się ze sobą kolejnością. W ten sposób dane osobniki dotykają małe mutacje punktowe.
Ogólny schemat działania metaheurystyki jest następujący. Najpierw z macierzy tworzona jest populacja początkowa (ten zabieg wykonywany jest tylko raz – na starcie). Kolejno algorytm bazując na tej populacji przez zadaną liczbę generacji liczy fitness każdego osobnika, zapamiętuje rozwiązanie, wybiera osobników do reprodukcji, krzyżuje je ze sobą, wprowadza mutacje i cykl jest powtarzany.

3. Testowanie

Celem zbadania skuteczności działania zaimplementowanej metaheurystyki przeprowadzono rozbudowane testy, w których badano wpływ zmiany jej parametrów na funkcję celu oraz czas obliczeń. Oprócz testowania parametrów metaheurystyki przeprowadzono również testy parametrów instancji na funkcję celu i czas obliczeń. Testy przeprowadzono na wielu instancjach różniących się rozmiarem, poziomem trudności i liczbą błędów. Każda instancja o danym rozmiarze, danej ilości wprowadzonych błędów i danej trudności była reprezentowana przez 10 losowo różniących się macierzy (o tych samych parametrach) i każdą z nich testowano w celu uśrednienia wyników. Zanim przystąpiono do pełnych testów na wszystkich instancjach przeprowadzono testy wstępne na podzbiorze tych instancji aby móc zawęzić zestaw możliwych parametrów metaheurystyki do tych najbardziej skutecznych względem rozmiaru instancji, a także aby móc wstępnie ocenić skuteczność działania metaheurystyki.

Specyfikacja urządzenia, na którym przeprowadzono testy:

- *Procesor: AMD Ryzen 5 4500U*
- *Pamięć RAM: 16GB*
- *Typ systemu: 64-bitowy system operacyjny, procesor x64*
- *System operacyjny: Windows 10 PRO*

3.1. Testowanie wstępne

Macierz o wymiarach 5x5

Na ogół algorytm bardzo szybko znajduje pożądane rozwiązanie, nierzadko wynajduje również uporządkowania kolumn o niższej wartości funkcji celu niż ta wyliczona dla rozwiązania. Dla tego rozmiaru macierzy zdecydowano się na stały parametr trudności instancji – maksymalna liczba jedynek w wierszu to 4, ponieważ nawet wtedy instancje nie stanowią większej trudności dla metaheurystyki. Testy wstępne wykazały, że dla tak prostych instancji największy wpływ na obliczenia wywierają parametry: rozmiar populacji oraz liczba iteracji i manipulacja tymi parametrami jest wystarczająca by osiągnąć rozwiązanie o docelowej wartości funkcji celu, a nawet niższej. Dlatego tez skupiono się tylko na nich testując te instancje. W tym przypadku badana liczba iteracji mieści się w przedziale od 3 do 9, a rozmiar populacji od 5 do 15.

Macierz o wymiarach 10x10

Są to już nieco trudniejsze instancje i zaobserwowano, że metaheurystyka rzadko w (w porównaniu do poprzednich instancji) zwraca rozwiązania o wartości funkcji celu równej wartości rozwiązania (lub niższej). Skuteczniejsze w tym wypadku były większe rozmiary populacji, dlatego zdecydowano żeby ograniczyć się do wartości ze zbioru od 50 do 350. Zauważono również, że wymagana jest zdecydowanie większa

liczba iteracji niż poprzednio – przedział od 10 do 30. Na pierwszy rzut oka nie zaobserwowano żadnych znaczących zmian wartości funkcji celu w zależności od manipulowania prawdopodobieństwem krzyżowania i mutacji, lecz parametry te zostały szerzej zbadane w testach dokładnych, w dodatku są to parametry, dla których w każdej instancji warto testować całe spektrum gdyż ich maksymalna wartość to 1. Jeśli chodzi o rodzaje instancji o tym rozmiarze, to generowano i badano instancje z poziomem trudności 4 oraz 8 i z ilością błędów 2, 5 i 8.

Macierz o wymiarach 15x15

Dla tego typu instancji zauważono już wzrost złożoności i wyniki zwracane nie znajdowały się już w bardzo bliskim sąsiedztwie wartości funkcji celu rozwiązania instancji. Brano już pod uwagę coraz większe rozmiary populacji – od ok. 200 do ok. 800, oraz coraz większe liczby iteracji: od ok. 20 aż do ok. 60. Badano również pozostałe 2 parametry. Generowano i testowano instancje o poziomach trudności 4, 7 i 11 z liczbami błędów: 3, 7, 11.

Macierz o wymiarach 20x20

Dla tego rozmiaru zaobserwowano, że metaheurystyka zwraca już dużo gorsze wyniki niż w przypadku poprzednich. Niezbędne było testowanie jeszcze większych wartości parametrów – rozmiaru populacji [320-950] i liczby iteracji [25-70]. Instancje o tym rozmiarze były o poziomach trudności 6 lub 12 z 4, 9 lub 14 błędami.

Dla wszystkich rozwiązań tworzonych instancji liczono również ich funkcję celu, i za każdym razem była to liczba równa liczbie wprowadzonych błędów.

Dla każdego badanego parametru tworzono dwa wykresy – wartości funkcji celu i czasu obliczeń w zależności od tego parametru. Aby zbadać zależność funkcji celu od czasu obliczeń wystarczy porównać wartości z osi y dla odpowiadających sobie punktów na dwóch wykresach. Wykresy tworzone były w języku *Python.*

Aby można było porównać wyniki metaheurystyki nie tylko z rozwiązaniami, ale i wartością funkcji celu rozwiązania losowego w tabeli niżej pokazano jakie wartości fitnessu otrzymywane są dla danych macierzy jako rozwiązania losowe (uśredniono 10 pomiarów dla każdego rozmiaru)

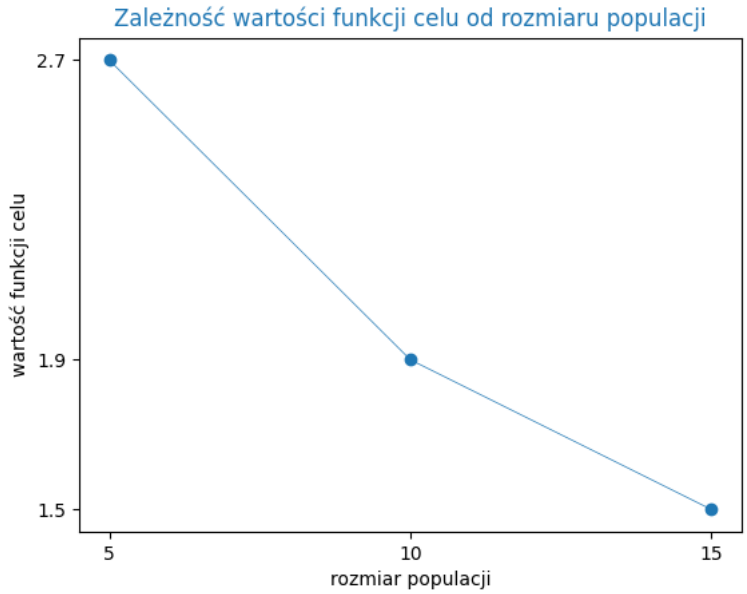
Rozmiar	5x5	10x10	15x15	20x20
Wartość funkcji celu rozwiązania losowego	4.9	15.1	56.8	83.2

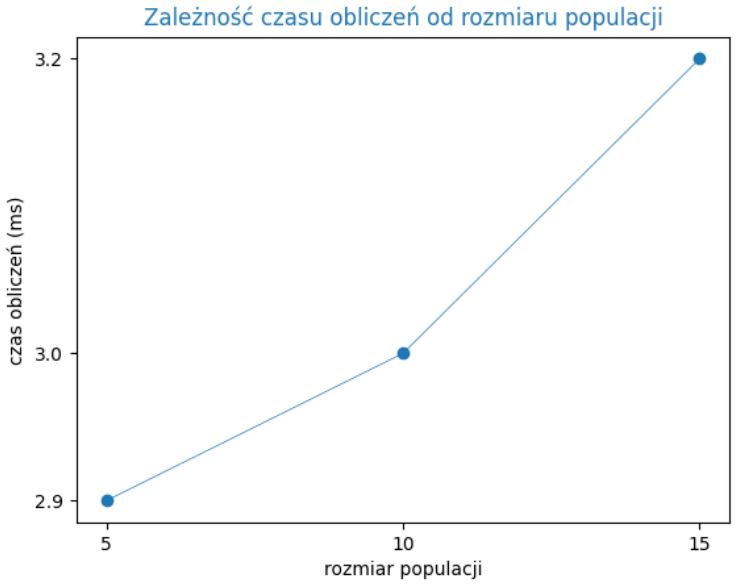
3.2. Testowanie dokładne metaheurystyki

Macierz o wymiarach 5x5 z poziomem trudności 4 oraz 2 błędami:

Badany parametr: **rozmiar populacji**.

Pozostałe parametry: liczba iteracji – 3, prawdopodobieństwo krzyżowania – 0.5, prawdopodobieństwo mutacji – 0.2

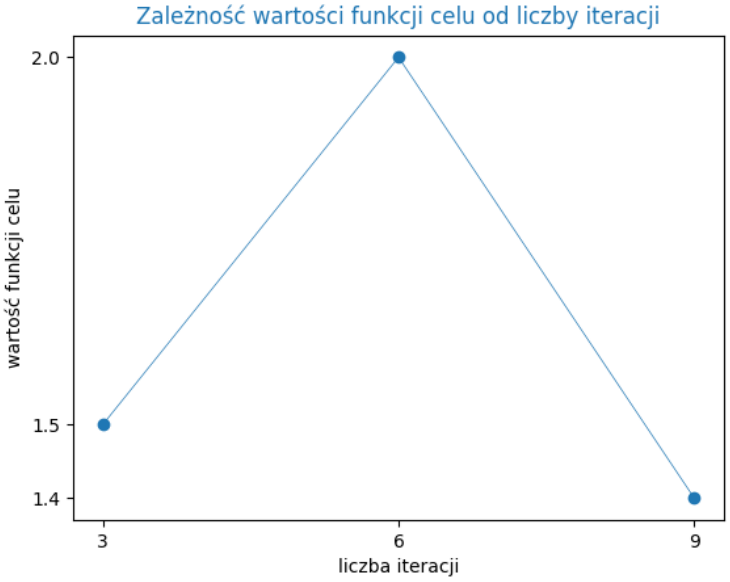


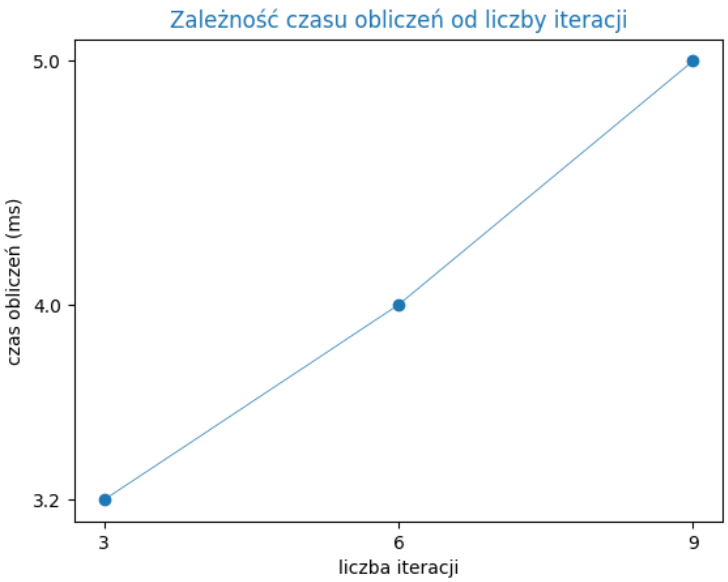


Wnioski:

Mimo niskiego poziomu złożoności i dobrych wyników dla każdej wartości rozmiaru populacji, można zaważyć, że zwiększanie tego parametru wpływa na obniżenie wartości funkcji celu. Podobnie zwiększanie rozmiaru populacji w małym stopniu, ale wpływa na wydłużenie czasu obliczeń. Czas mierzony jest w milisekundach, więc są to niezauważalne różnice.

Badany parametr: **liczba iteracji**.
Pozostałe parametry: rozmiar populacji - 15, prawdopodobieństwo krzyżowania – 0.5, prawdopodobieństwo mutacji – 0.2





Wnioski:
Ponownie wykazano, że czas obliczeń zwiększa się wraz ze wzrostem iteracji, jednak tutaj ten wpływ jest dużo większy niż w przypadku poprzedniego parametru. Wykres zależności wartości funkcji celu od liczby iteracji pokazuje, że w tak małej instancji duże znaczenie ma czynnik losowy, a dokładniej w porównaniu z poprzednim parametrem, że rozmiar populacji w której generowane są losowe osobniki ma większe znaczenie.

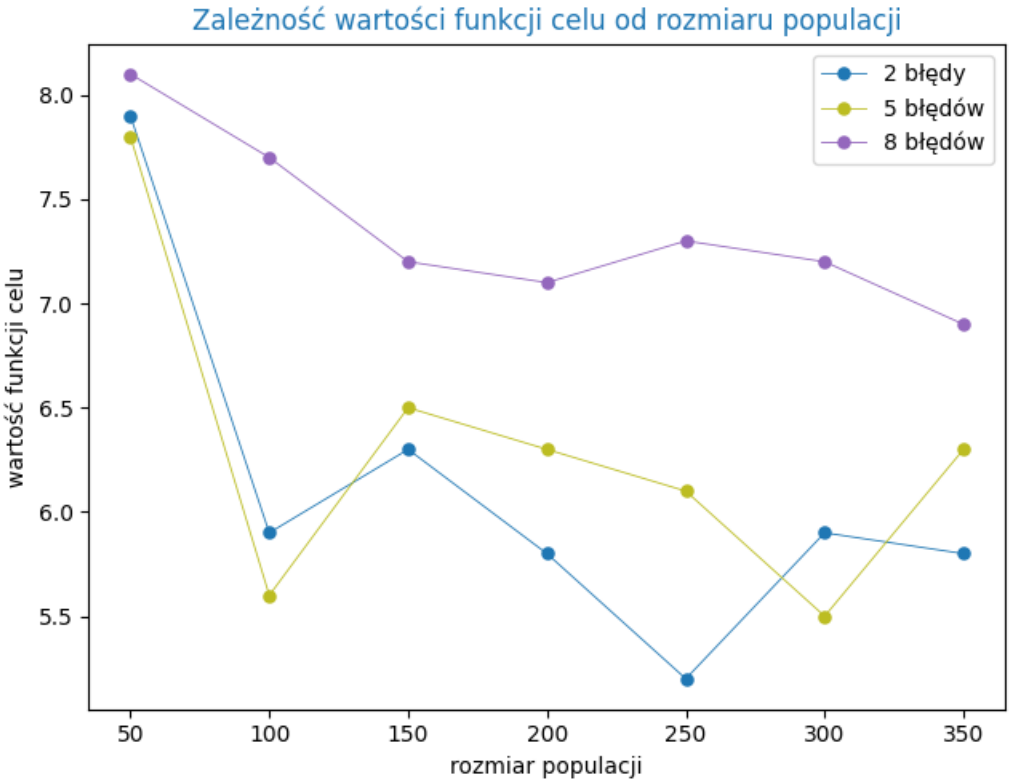
Macierz o wymiarach 5x5 z poziomem trudności 4 oraz 4 błędami:

Zauważono, że dla tak małej instancji wprowadzenie większej ilości błędów nie wpływa istotnie na wartość funkcji celu czy czas obliczeń. Oczywiście w tym wypadku liczba błędów różni się tylko o 2 co nie jest dużą różnicą, ale w tak małej instancji jest pewna maksymalna liczba możliwych do wprowadzenia błędów - ok. 6 (ponieważ jest to mocno specyficzne w zależności od ułożenia zer i jedynek). Dlatego zdecydowano się na zaprzestanie dalszego badania wpływu zwiększenia ilości błędów na działanie metaheurystyki na tych instancjach. Porównano tylko część wyników uzyskanych dla instancji z 2 błędami.
Badany parametr: **wpływ liczby błędów dla małej, prostej instancji**.
Pozostałe parametry: rozmiar populacji - 15, prawdopodobieństwo krzyżowania – 0.5, prawdopodobieństwo mutacji – 0.2, liczba iteracji - 3

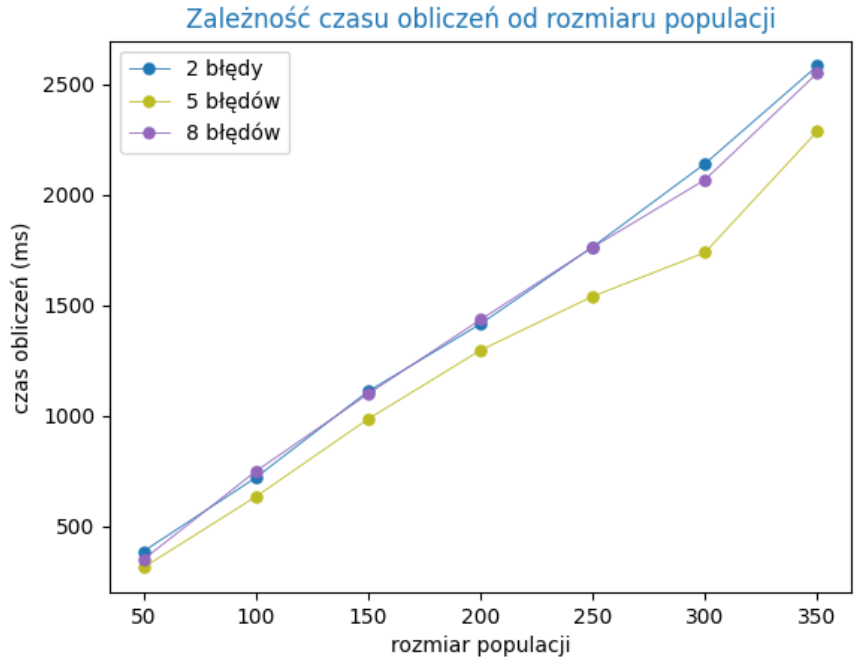
Instancja z 2 błędami		Instancja z 4 błędami	
Czas obliczeń (ms)	Wartość f celu	Czas obliczeń (ms)	Wartość f celu
3.2	1.5	3.6	2.3

Wnioski:
Widoczna jest różnica zarówno w czasie obliczeń jak i wartości funkcji celu, ale są to w rzeczywistości małe różnice. Na ogół dla tak małych instancji niezależnie od ilości błędów metaheurystyka wynajduje bardzo dobre rozwiązania, dlatego nie testowano już parametrów metaheurystyki na instancjach rozmiaru 5x5 ze zwiększającą się liczbą błędów.

Macierz o wymiarach 10x10 z poziomem trudności 4 oraz 2/5/8 błędami:
Badany parametr: **rozmiar populacji.**
Pozostałe parametry: liczba iteracji - 10, prawdopodobieństwo krzyżowania – 0.5, prawdopodobieństwo mutacji – 0.2



Wnioski:
Zaobserwować można istotny wpływ rozmiaru populacji na wartość funkcji celu.
Im większy ten parametr, tym wartość funkcji celu jest coraz niższa. Po przekroczeniu rozmiaru 250 metaheurystyka zwraca bardzo zbliżone wyniki (poza instancją z 5 błędami, ponieważ tutaj rozwiązania przy rozmiarze populacji 250 są wyjątkowo słabe). Zwiększanie tego parametru wywiera zdecydowanie mniejszy wpływ na instancje z 8 błędami.

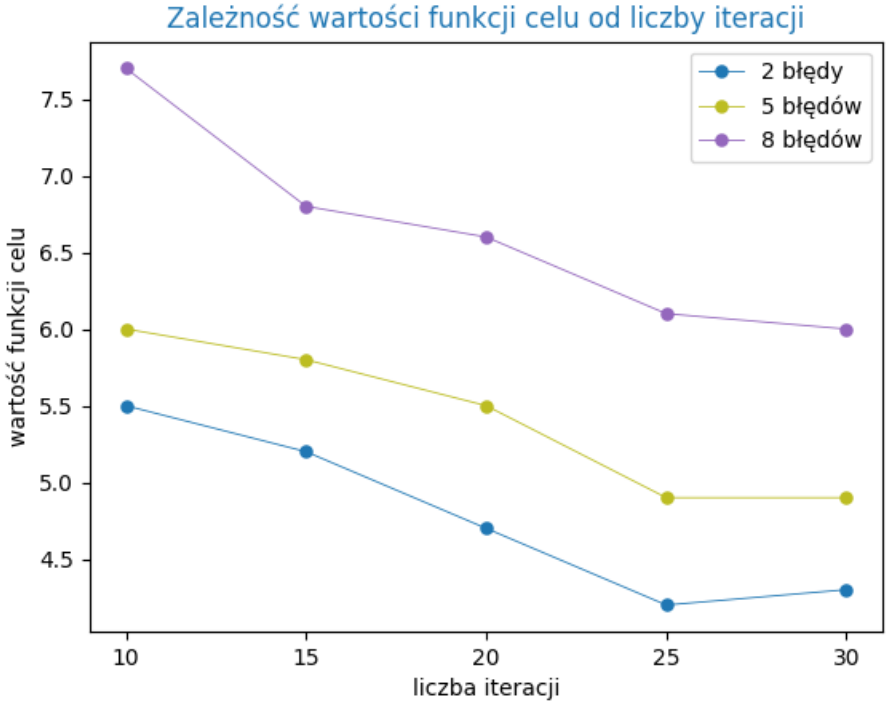


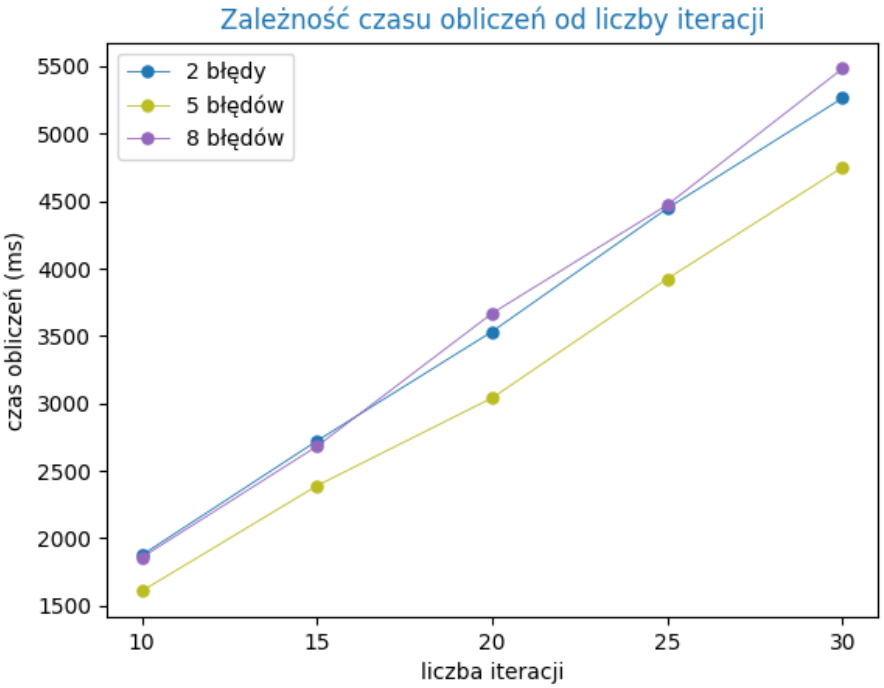
Wnioski:

Czas znacząco rośnie w miarę wzrostu populacji, niemalże wprost proporcjonalnie. Liczba błędów instancji nie ma tu natomiast żadnego znaczenia. Widać, że instancje 10x10 są dużo bardziej złożone niż instancje 5x5. W tych drugich czas liczony był w milisekundach, a dla macierzy 10x10 są już to sekundy. Dodatkowo metaheurystyka dużo rzadziej, jeśli w ogóle, zwracała wynik równy rozwiązaniu instancji lub niższy.

Badany parametr: **liczba iteracji.**

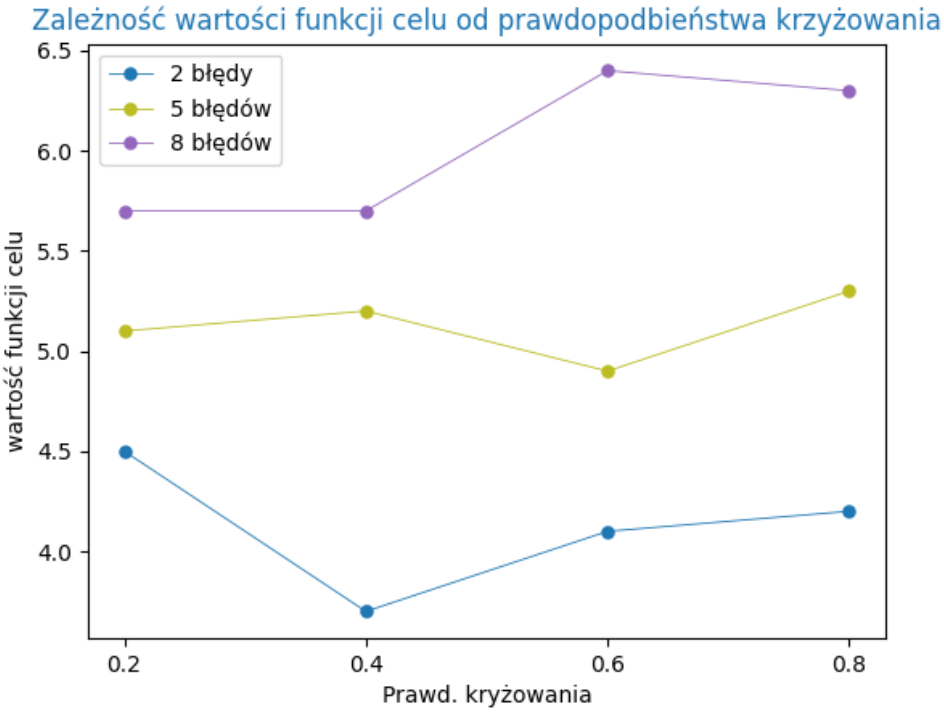
Pozostałe parametry: rozmiar populacji - 250, prawdopodobieństwo krzyżowania – 0.5, prawdopodobieństwo mutacji – 0.2

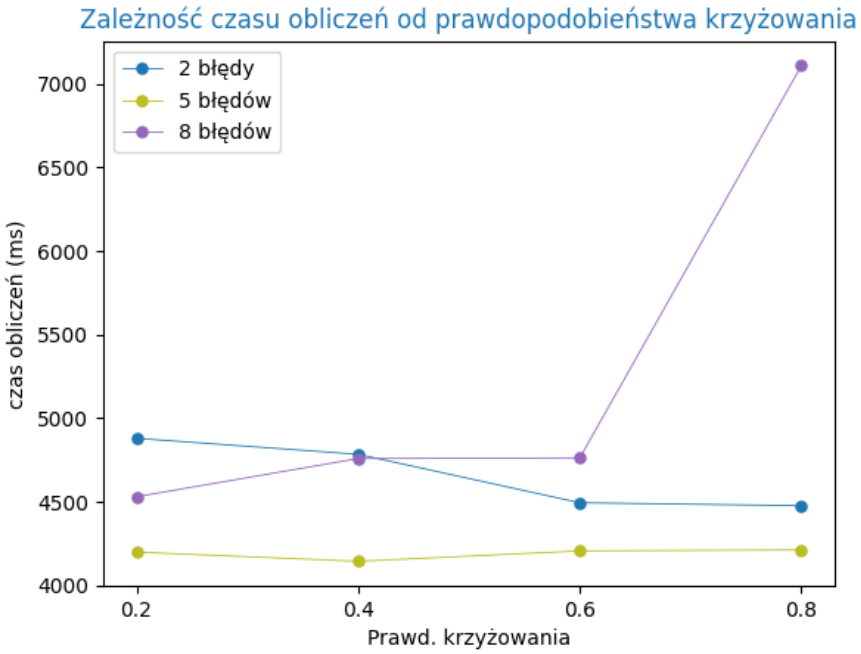




Wnioski:
Liczba iteracji zdecydowanie powinna być wyższa niż niższa z badanego zbioru. Liczba iteracji 25 jest wystarczająca dla instancji tego rodzaju. Widać, że zwiększanie tego parametru wywiera porównywalny wpływ na instancje z różną liczbą błędów. Czas ponownie znacząco rośnie w miarę zwiększania parametru.

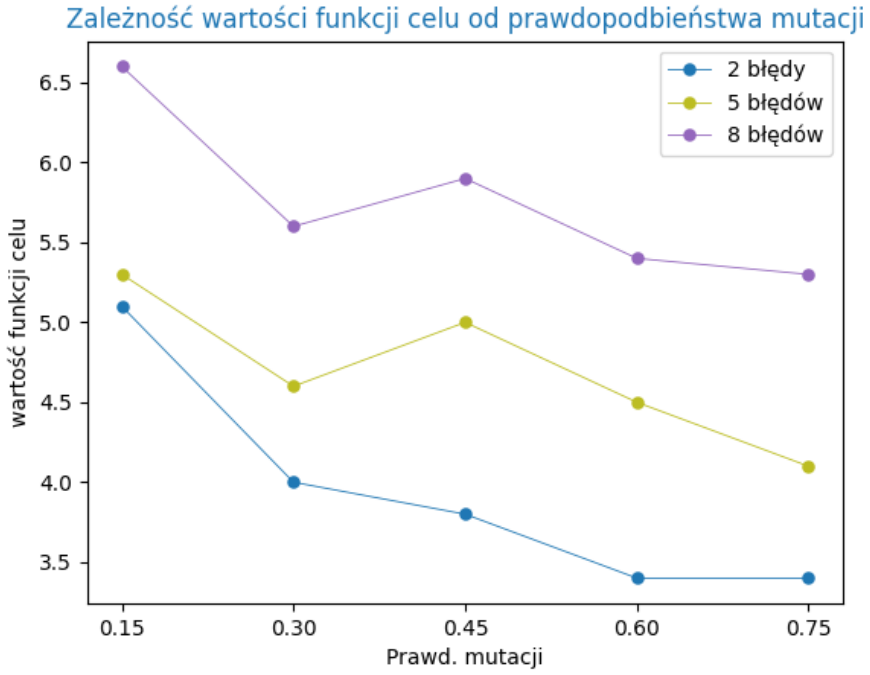
Badany parametr: **prawdopodobieństwo krzyżowania.**
Pozostałe parametry: rozmiar populacji - 250, liczba iteracji – 25, prawdopodobieństwo mutacji – 0.2

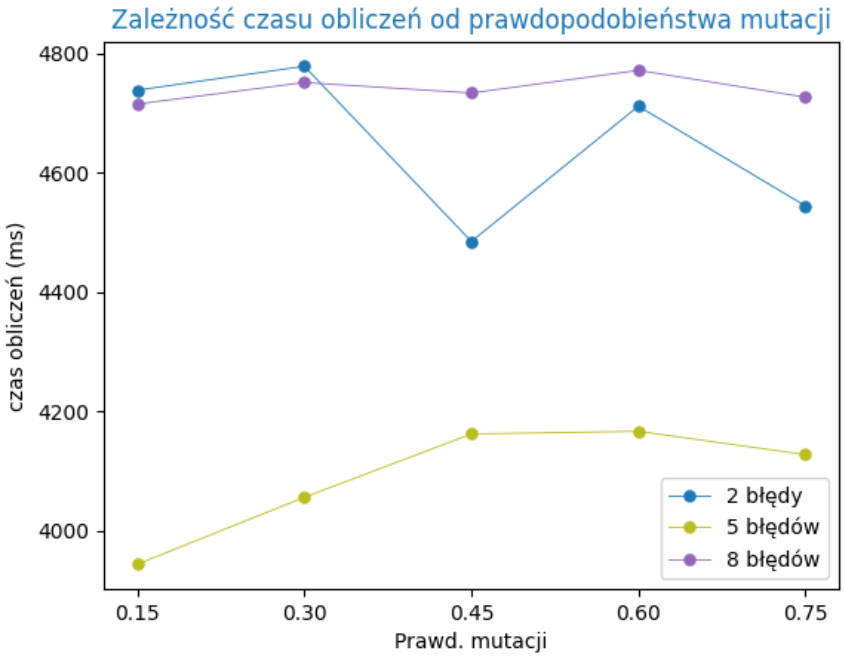




Wnioski:
Prawdopodobieństwo krzyżowania powinno być w zakresie 0.4 – 0.5, choć w badanych instancjach nie ma ono zbyt dużego znaczenia. Jednak można stwierdzić, że zbyt duże prawdopodobieństwo np. 0.8 zdecydowanie pogarsza rozwiązania. Zmiana tego parametru nie wpływa znacząco na czas obliczeń. Dopiero w instancji z 8 błędami zauważono nienaturalne wydłużenie czasu obliczeń przy większych wartościach badanego parametru.

Badany parametr: **prawdopodobieństwo mutacji.**
Pozostałe parametry: rozmiar populacji - 250, liczba iteracji – 25, prawdopodobieństwo krzyżowania– 0.4





Wnioski:

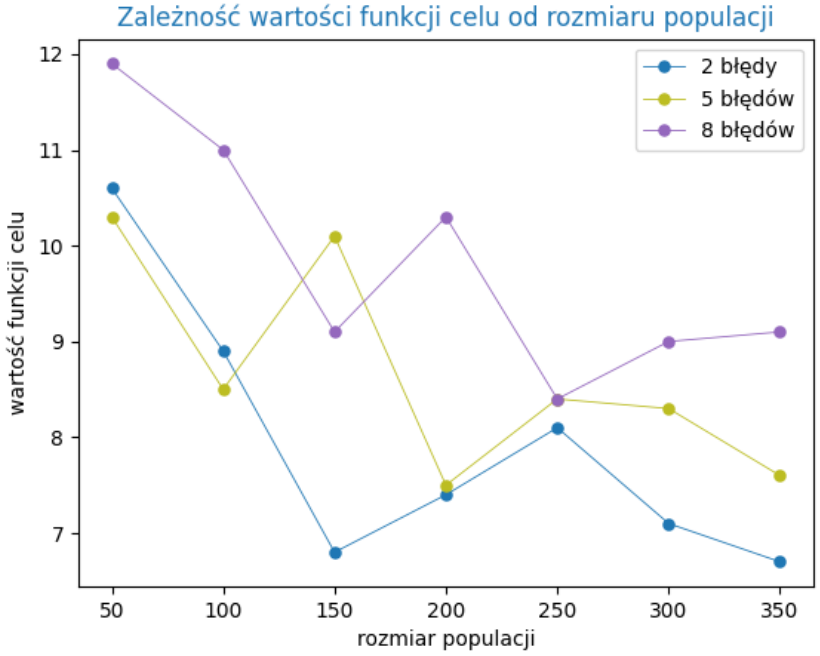
Kiedy prawdopodobieństwo mutacji osiąga wyższe wartości to średnie wartości funkcji celu są niższe dla instancji ze wszystkimi ilościami błędów. W tym wypadku wartość 0.6 – 0.7 wydaje się być najbardziej skuteczna. Z założenia mutacja to zjawisko dosyć rzadkie wiec otrzymany wynik trochę kłóci się z teorią.

Zmiana tego parametru nie wpływa znacząco na czas obliczeń. Różnice na wykresie wynikają raczej z losowości i są nieznaczne.

Macierz o wymiarach 10x10 z poziomem trudności 8 oraz 2/5/8 błędami:

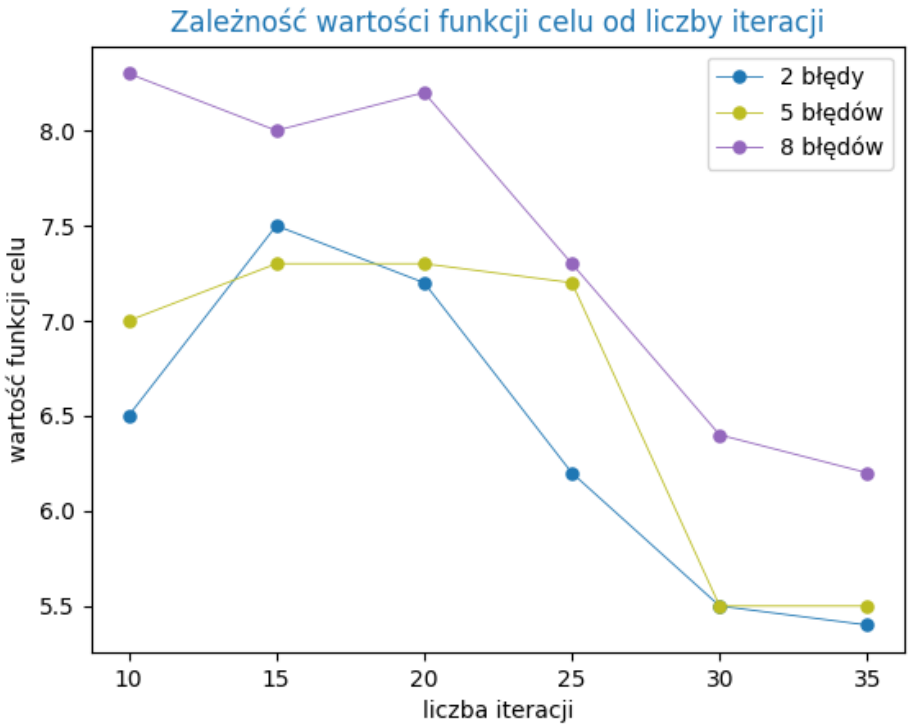
Badany parametr: **rozmiar populacji.**

Pozostałe parametry: liczba iteracji – 10, prawdopodobieństwo krzyżowania – 0.5 prawdopodobieństwo mutacji – 0.2



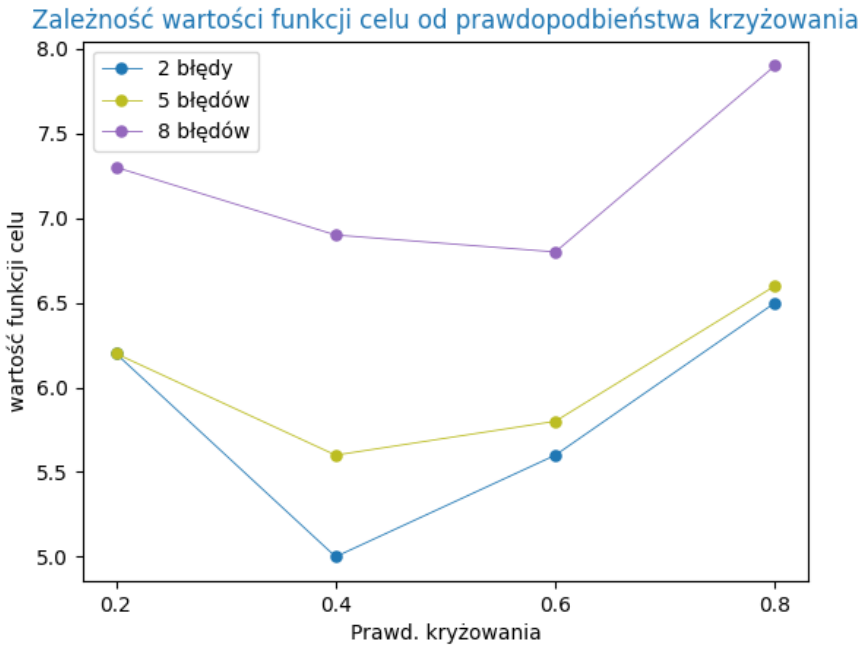
Wnioski:
Mimo, że nadal widać zdecydowaną korzyść z zastosowania wyższych wartości parametru rozmiaru populacji, to widać że w instancjach o większym stopniu trudności dużo większe znaczenie ma przypadek. Nie sposób znaleźć jedną najbardziej optymalną wartość dla instancji o różnych odsetkach błędów(jak udawało się to w przypadku testowania instancji o niższym poziomie trudności), dlatego podczas testowania dalszych parametrów można by przyjąć 150 jako liczbę populacji dla instancji z 2 błędami, 200 dla 5 błędów i 250 dla 8 błędów. Jednak zakres pozostaje dość wąski 150 – 250 i nie ma sensu redukowania parametru dla niektórych instancji. Różnice na wykresie są niewielkie i wynikają z przypadku, a koszt czasowy między rozmiarem 150 a 250 nie jest znaczący. Utrzymanie tego parametru na poziomie 250 może skutkować dużo lepszymi wynikami w kolejnych testach.
Nie tworzono już wykresów wpływu zmiany parametru na czas obliczeń, ponieważ jest to analogiczne do poprzednich testów, jedynie czasy były nieco wyższe. Wpływ trudności instancji na czas obliczeń/funkcję celu będzie zobrazowany w dalszej części sprawozdania przy testowaniu parametrów instancji.

Badany parametr: **liczba iteracji.**
Pozostałe parametry: rozmiar populacji –250, prawdopodobieństwo krzyżowania – 0.5 prawdopodobieństwo mutacji – 0.2



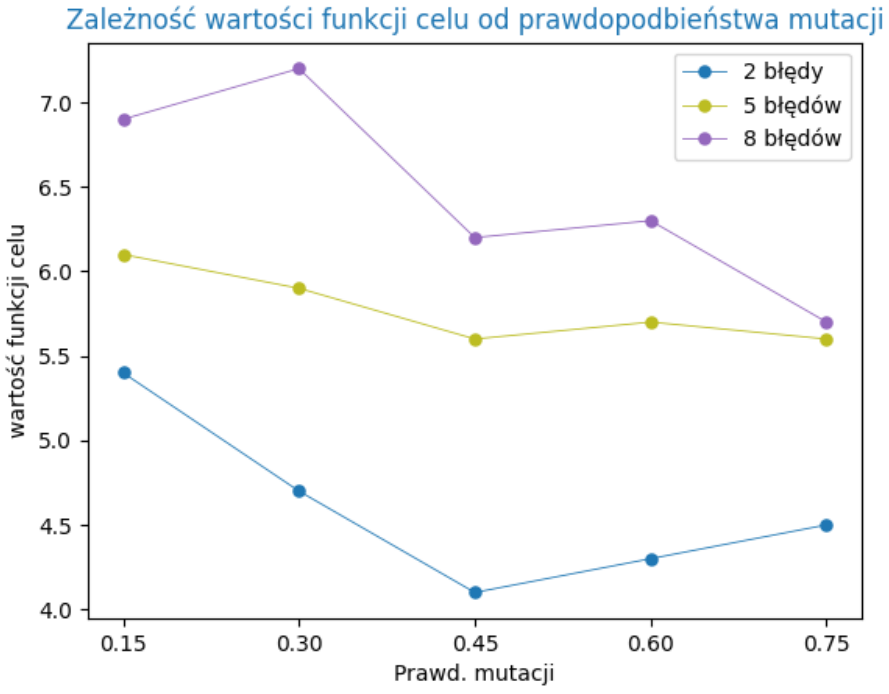
Wnioski:
Widać poprawę w zależności od wzrostu liczby iteracji.
W trakcie testów zdecydowano się zwiększyć jeszcze liczbę iteracji o 5 w porównaniu z testowaniem instancji o mniejszym poziomie trudności. Wybrano jednak parametr 30 Iteracji do dalszych testów.

Badany parametr: **prawdopodobieństwo krzyżowania.**
Pozostałe parametry: rozmiar populacji –250, liczba iteracji – 30, prawdopodobieństwo mutacji – 0.2



Wnioski:
Ponownie prawdopodobieństwo krzyżowania w okolicach ½ (0.4-0.6) zdaje się być najbardziej skuteczne.

Badany parametr: **prawdopodobieństwo mutacji.**
Pozostałe parametry: rozmiar populacji –250, liczba iteracji – 30, prawdopodobieństwo krzyżowania – 0.4



Podsumowanie testowania instancji o rozmiarze 10x10

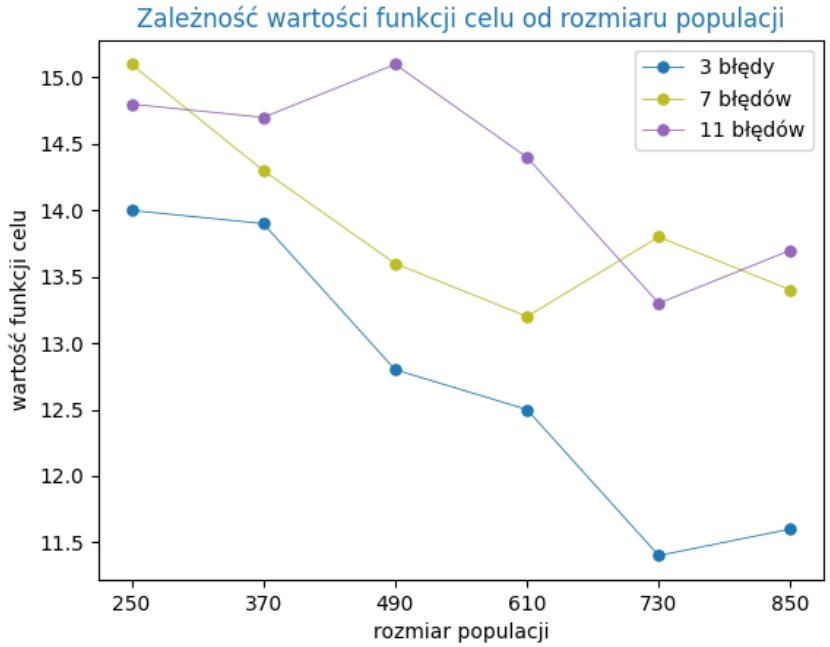
Poziom trudności	4			8		
Liczba błędów	2	5	8	2	5	8
f. celu czas	3.4 4544.8 ms	4.1 4127.8 ms	5.3 4727.2 ms	4.1 5947.4 ms	5.6 5776.8	6.2 5396.7

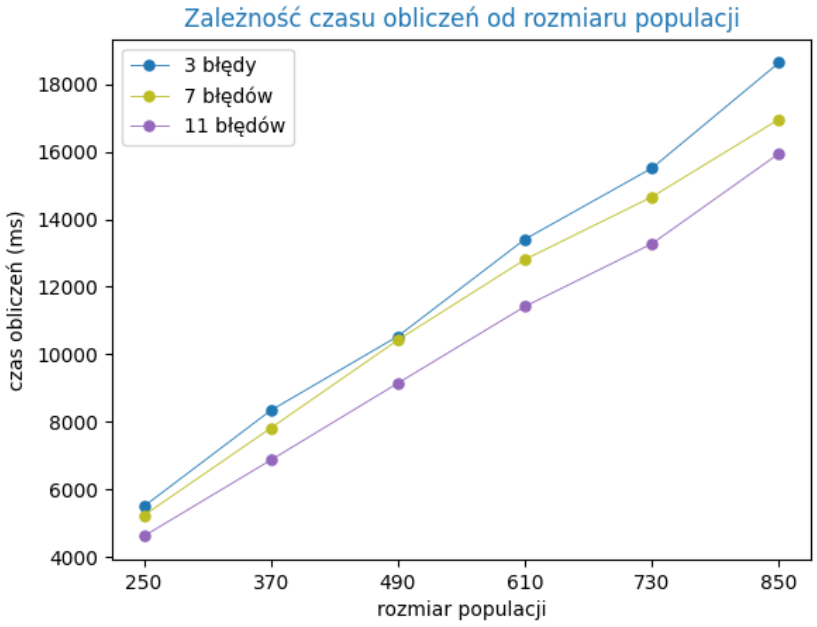
*podane najlepsze uzyskane wyniki i odpowiadający im czas.

Macierz o wymiarach 15x15 z poziomem trudności 4 oraz 3/7/11 błędami:

Badany parametr: **rozmiar populacji.**

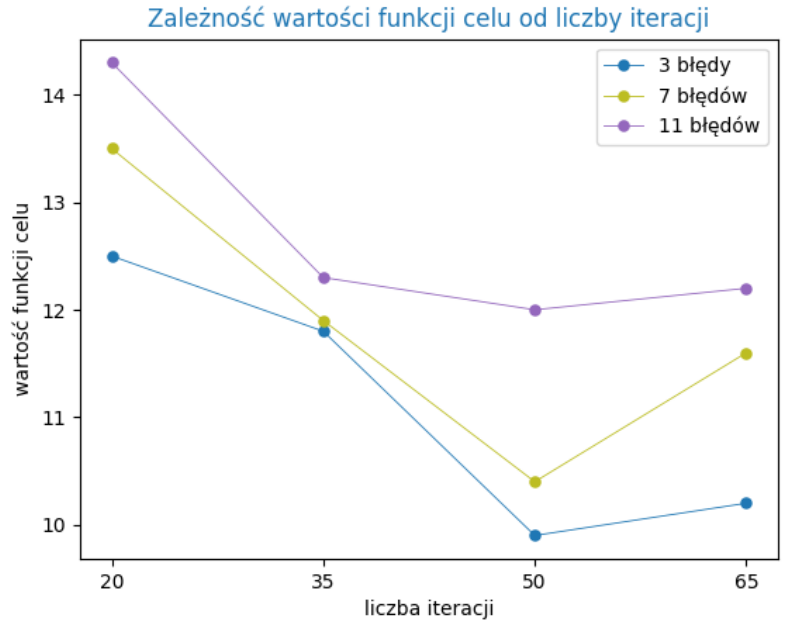
Pozostałe parametry: liczba iteracji – 15, prawdopodobieństwo krzyżowania – 0.5 prawdopodobieństwo mutacji – 0.2

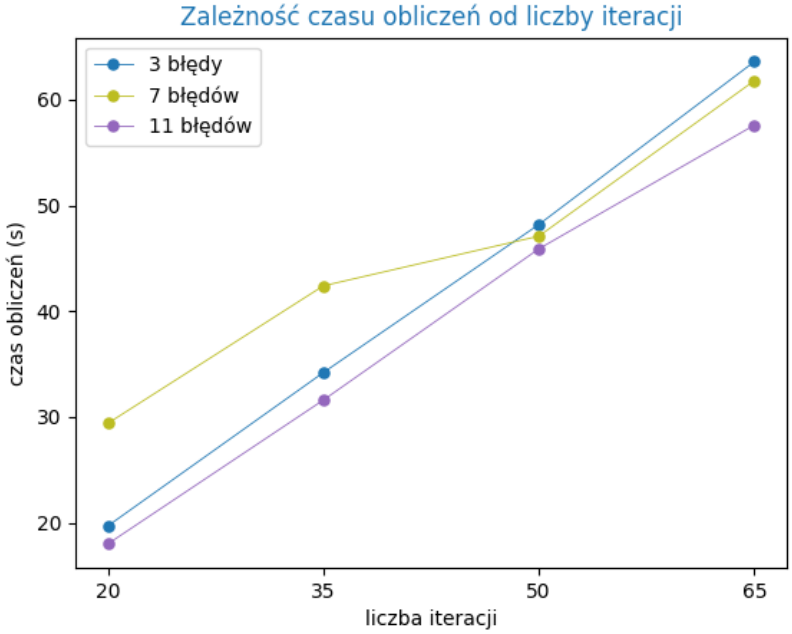




Wnioski:
Widać, że wyniki funkcji celu są już słabsze niż dla macierzy 10x10. Dla instancji z 3 błędami i z rozwiązaniem o funkcji celu = 3, udało się znaleźć jedynie rozwiązanie o funkcji celu 11.5. Do kolejnych testów wybrano rozmiar populacji 730. Widać również, że podobnie jak w mniejszych instancjach czas wzrasta proporcjonalnie ale osiąga już dużo większe wartości.

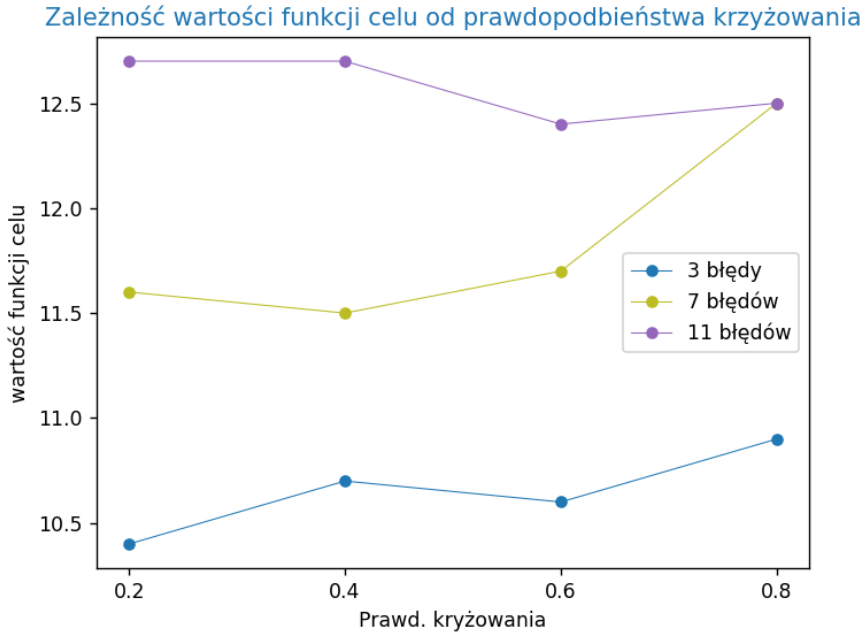
Badany parametr: **liczba iteracji.**
Pozostałe parametry: rozmiar populacji – 730, prawdopodobieństwo krzyżowania – 0.5 prawdopodobieństwo mutacji – 0.2

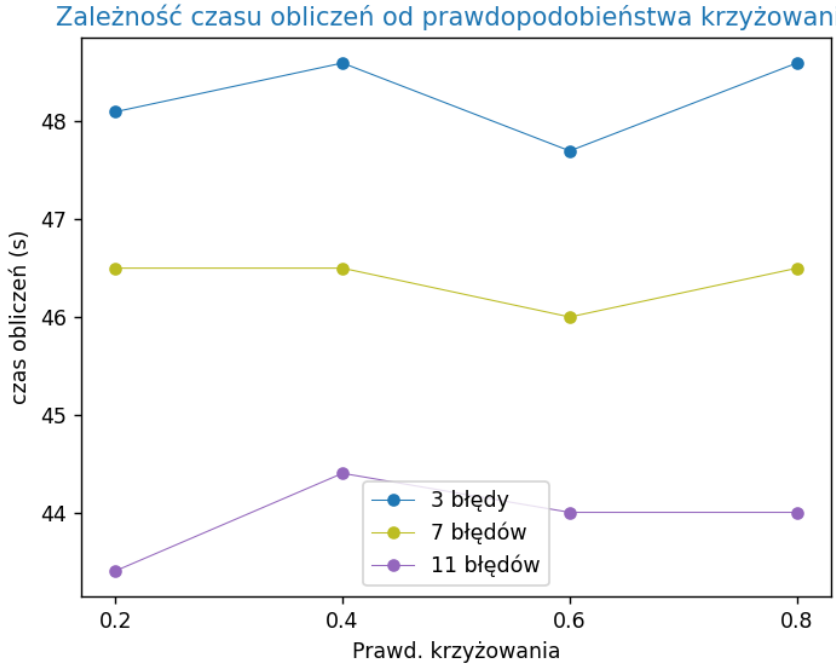




Wnioski:
Wybrana liczba iteracji dla instancji z każdą ilością błędów to 50. Udało się nieco zredukować różnicę między rozwiązaniami optymalnymi instancji, a rozwiązaniami zwróconymi przez metaheurystykę.

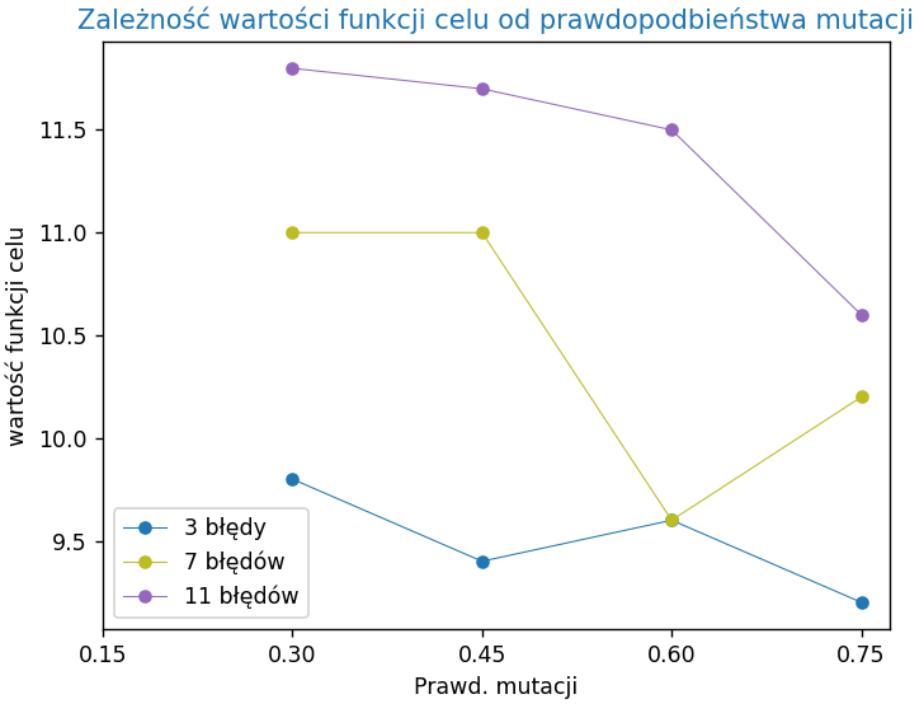
Badany parametr: **prawdopodobieństwo krzyżowania.**
Pozostałe parametry: rozmiar populacji – 730, liczba iteracji – 50, prawdopodobieństwo mutacji – 0.2





Wnioski:
Ponownie prawdopodobieństwo około połowy jest najkorzystniejsze, w tym wypadku przesunięte bardziej w stronę wartości 0.6. Mimo wszystko różnice jakie wywiera zmiana tego parametru są niewielkie. Brak wpływu zmiany tego parametru na czas obliczeń. Do dalszych testów wybrano parametr 0.6.

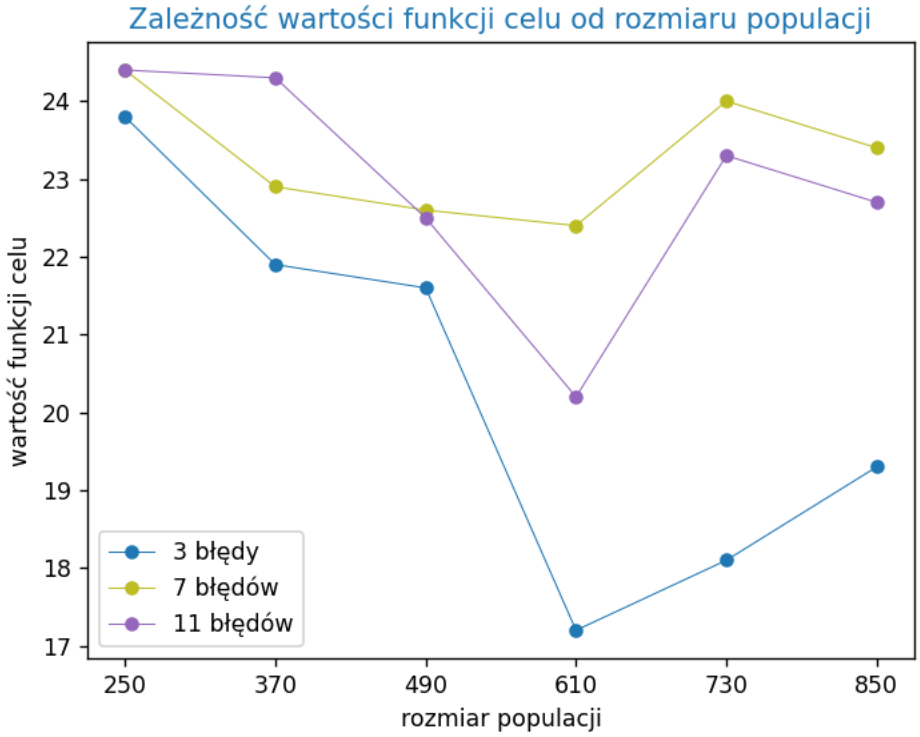
Badany parametr: **prawdopodobieństwo mutacji.**
Pozostałe parametry: rozmiar populacji – 730, liczba iteracji – 50, prawdopodobieństwo krzyżowania – 0.6

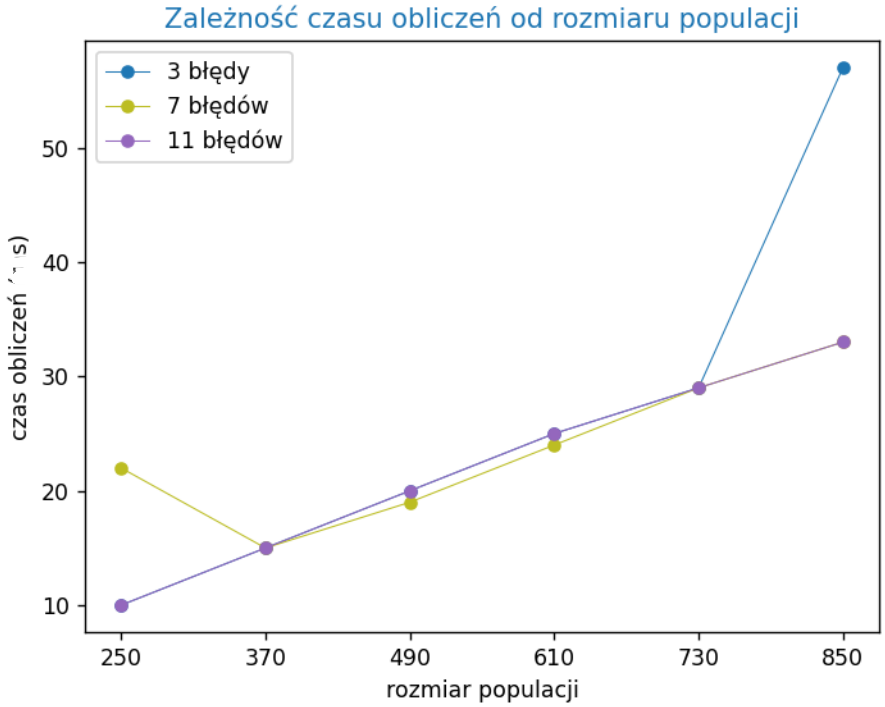


Wnioski:
Wysokie parametry prawdopodobieństwa mutacji pomogły ostatecznie trochę jeszcze obniżyć osiągane wartości funkcji celu.

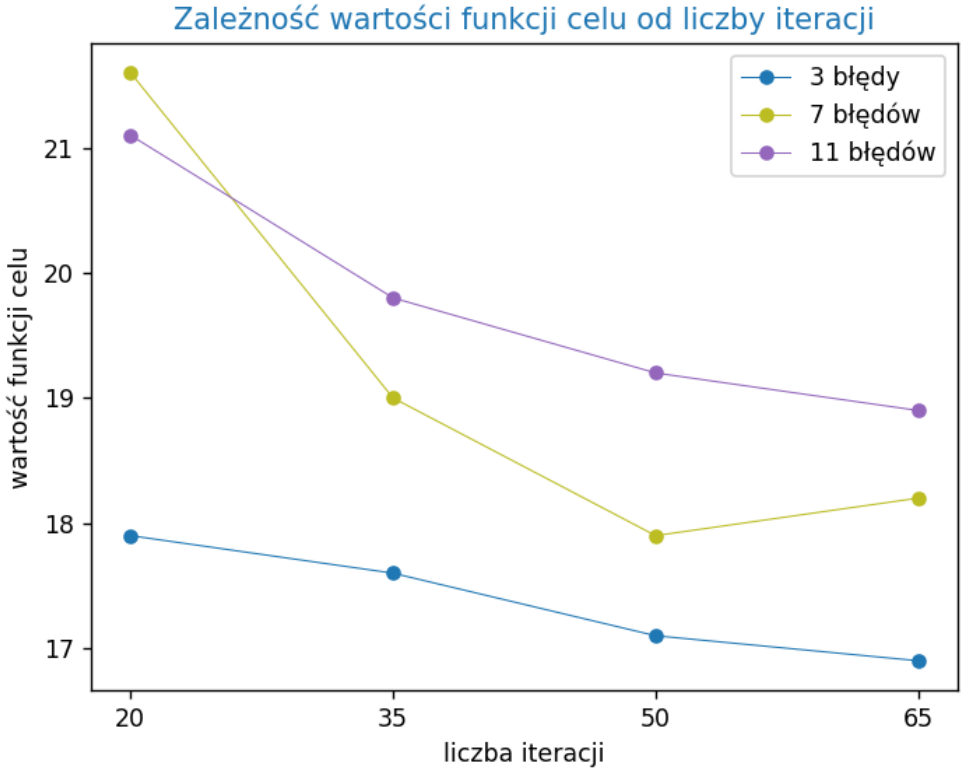
Macierz o wymiarach 15x15 z poziomem trudności 7 oraz 3/7/11 błędami:

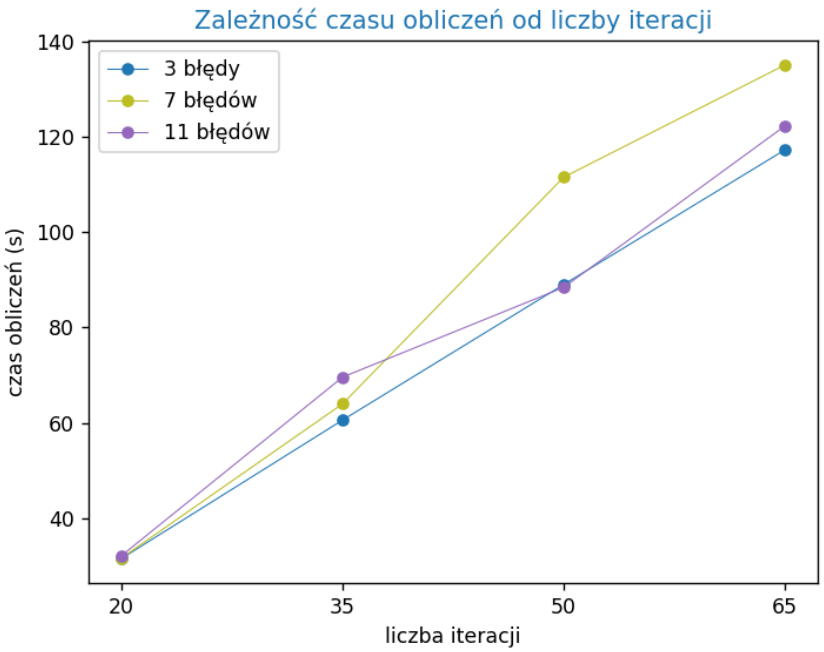
Badany parametr: **rozmiar populacji.**
Pozostałe parametry: liczba iteracji – 15, prawdopodobieństwo krzyżowania – 0.5 prawdopodobieństwo mutacji – 0.2





Wnioski: Najkorzystniej wypadła wartość 610 jako rozmiar populacji.
Zaobserwowano nienaturalny wzrost czasu obliczeń dla instancji z 3 błędami przy maksymalnej liczbie iteracji.
Badany parametr: **liczba iteracji.**
Pozostałe parametry: rozmiar populacji - 610, prawdopodobieństwo krzyżowania – 0.5 prawdopodobieństwo mutacji – 0.2



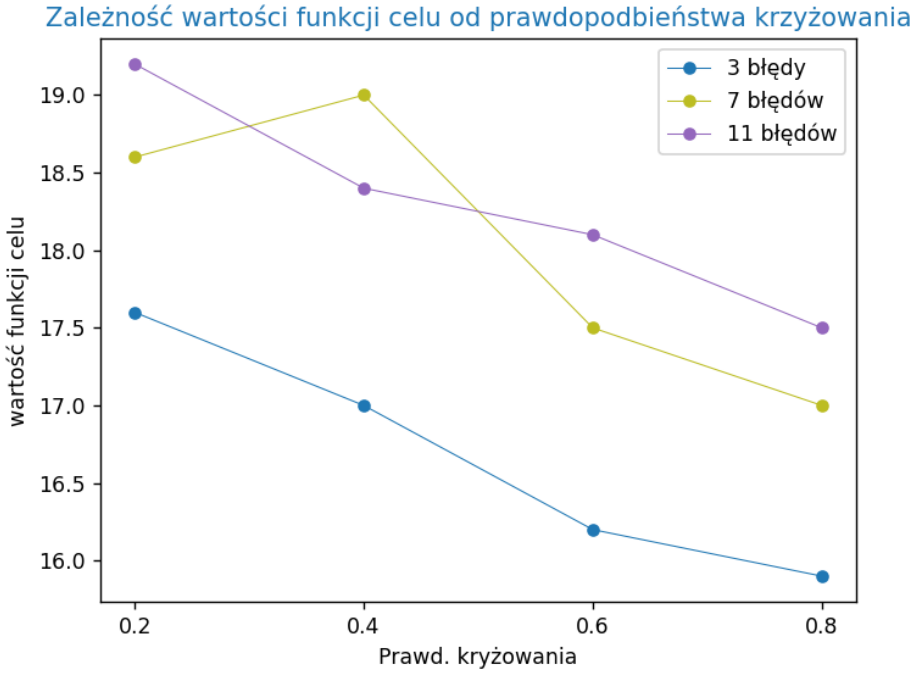


Wnioski:

Ponownie wybrano liczbę iteracji 50 do dalszych testów. Czas obliczeń zwiększa się coraz bardziej. Dla liczby 50 iteracji jest już to przeważnie ok. 80-120 sekund. Jak już wcześniej stwierdzono pozostałe parametry nie wpływają na czas obliczeń, zatem wyniki będą zobrazowane tylko dla funkcji celu. Czas zostanie podsumowany również w tabeli podsumowującej wyniki uzyskane dla instancji 15x15.

Badany parametr: **prawdopodobieństwo krzyżowania.**

Pozostałe parametry: rozmiar populacji - 610, liczba iteracji – 50 prawdopodobieństwo mutacji – 0.2



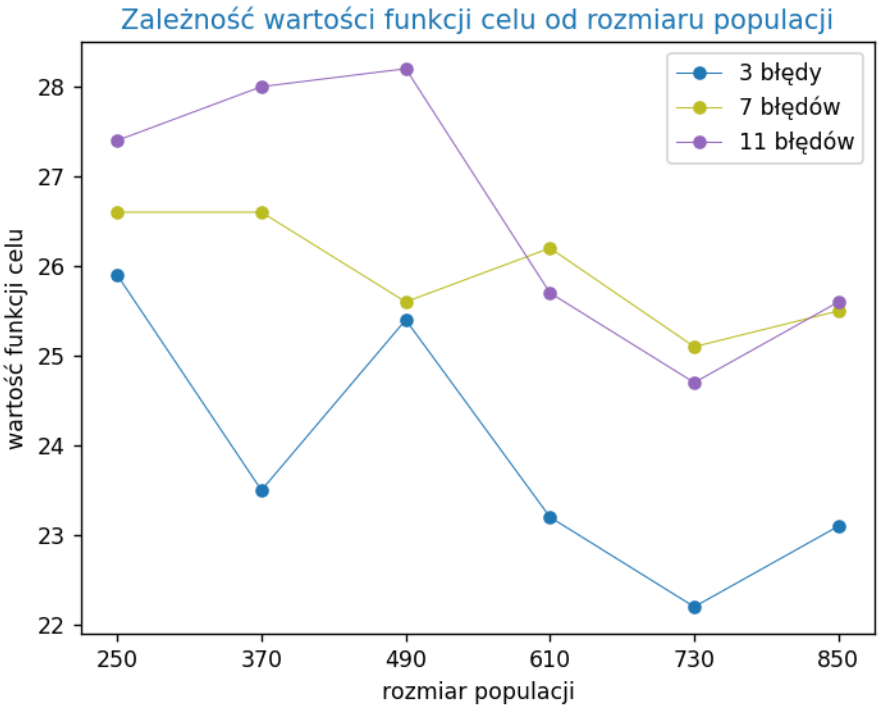
Wnioski:

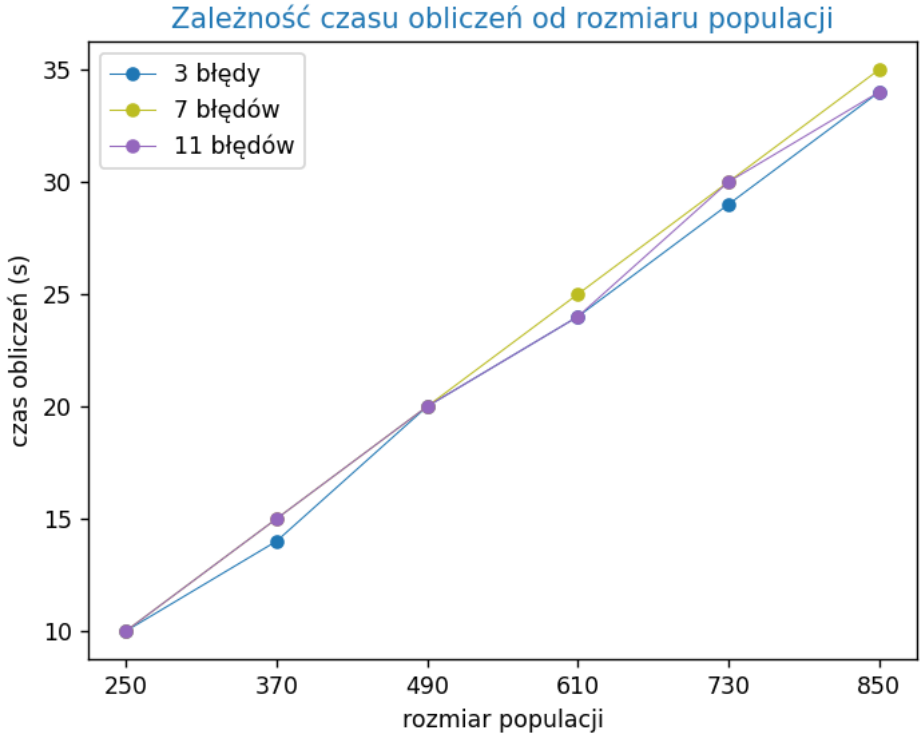
Najkorzystniej wypadają wartości prawdopodobieństwa krzyżowania między 0.6 a 0.8.

Macierz o wymiarach 15x15 z poziomem trudności 11 oraz 3/7/11 błędami:

Badany parametr: **rozmiar populacji.**

Pozostałe parametry: liczba iteracji – 15, prawdopodobieństwo krzyżowania – 0.5 prawdopodobieństwo mutacji – 0.2

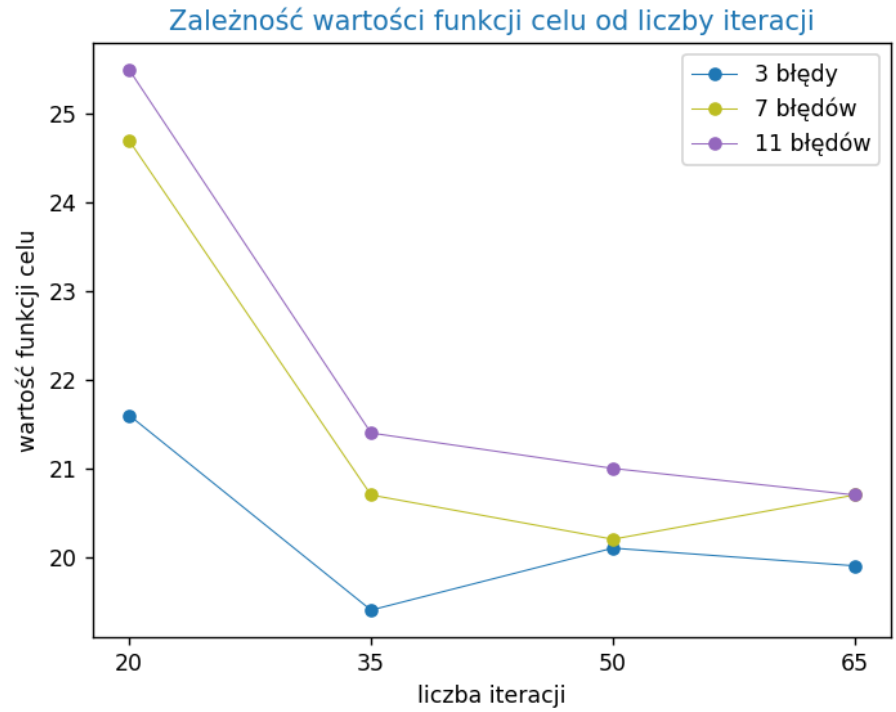


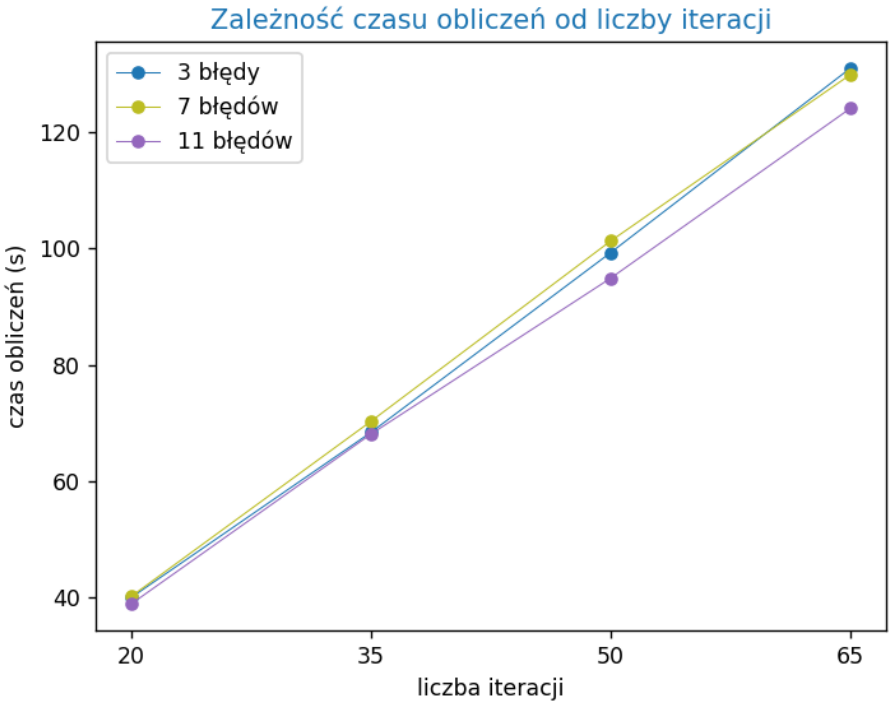


Wnioski:

Wybrany rozmiar populacji do dalszych testów to 730. Czas ponownie wzrasta wprost proporcjonalnie.

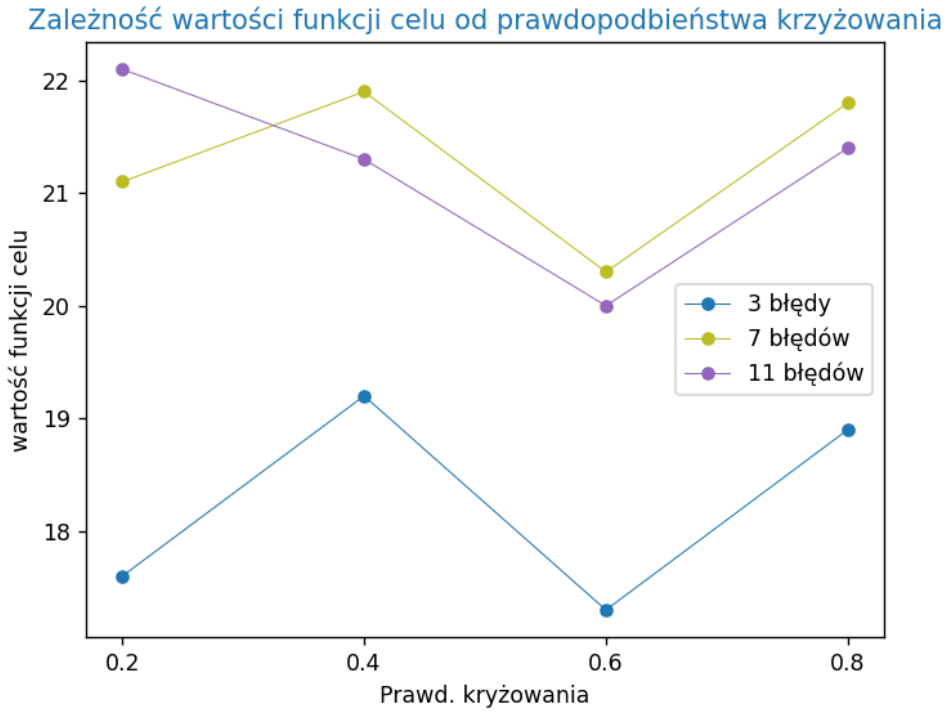
Badany parametr: **liczba iteracji.**
Pozostałe parametry: rozmiar populacji – 730, prawdopodobieństwo krzyżowania – 0.5 prawdopodobieństwo mutacji – 0.2





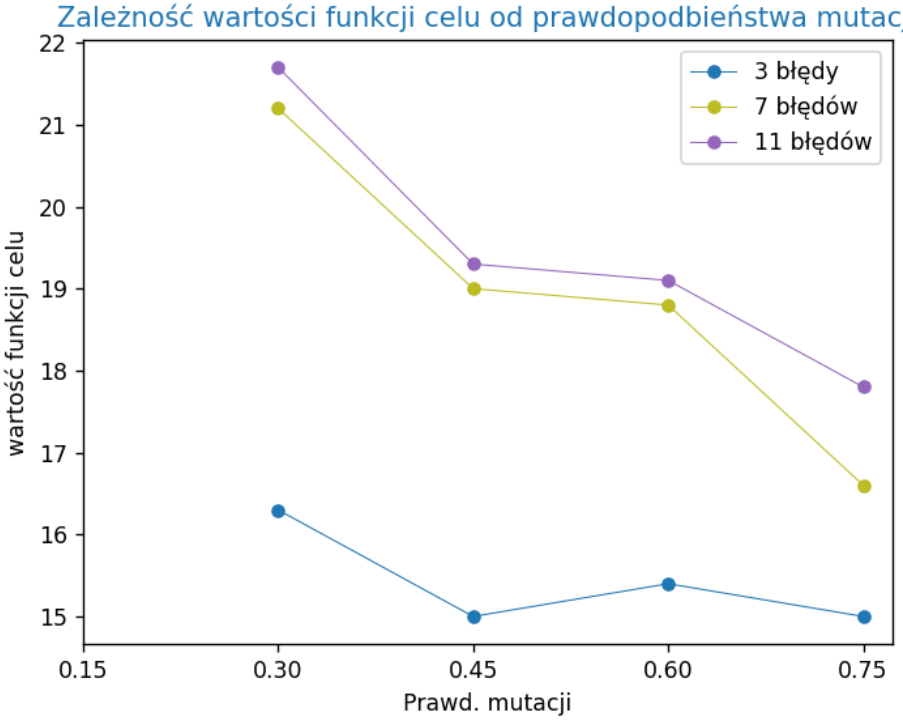
Wnioski:
Widoczna duża różnica w wartości funkcji celu między 20 a większą liczbą iteracji. W tym przypadku zdecydowanie wystarczająca jest również liczba 50 iteracji.

Badany parametr: **prawdopodobieństwo krzyżowania.**
Pozostałe parametry: rozmiar populacji – 730, liczba iteracji – 50, prawdopodobieństwo mutacji – 0.2



Wnioski:
Prawdopodobieństwo na poziomie 0.6 jest zdecydowanie najkorzystniejsze.

Badany parametr: **prawdopodobieństwo mutacji.**
Pozostałe parametry: rozmiar populacji – 730, liczba iteracji – 50, prawdopodobieństwo krzyżowania – 0.6



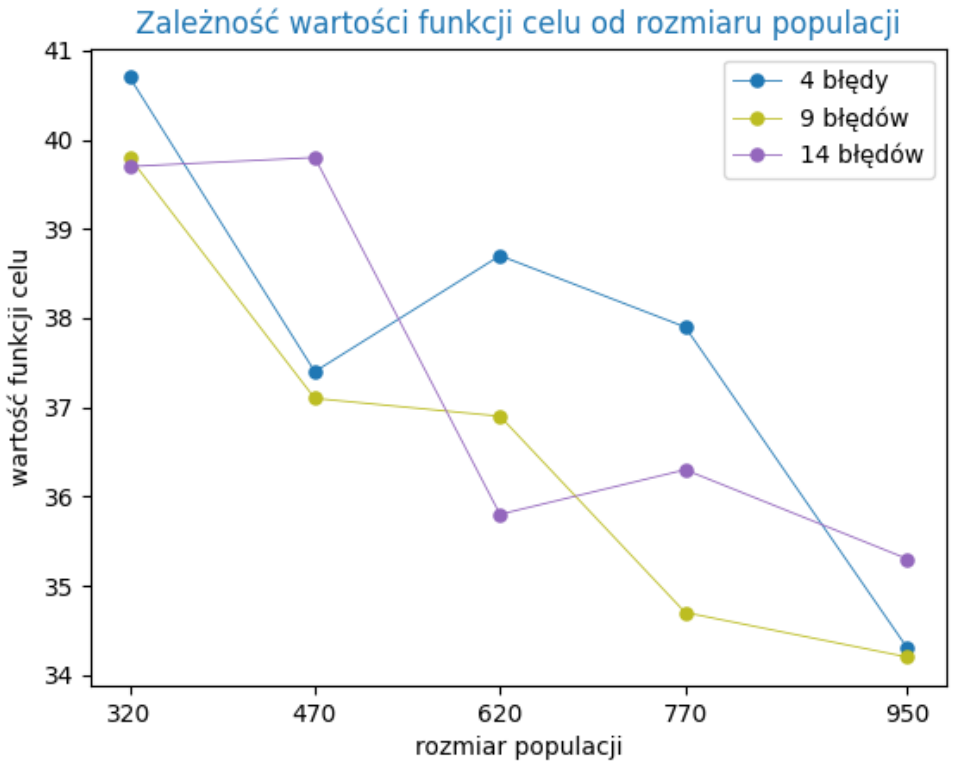
Wnioski:
Ponownie najlepsze wyniki osiągane są dla najwyższej wartości parametru. Ciekawa jest bardzo duża poprawa dla instancji z 7 i 11 błędami.

Podsumowanie testowania instancji o rozmiarze 15x15

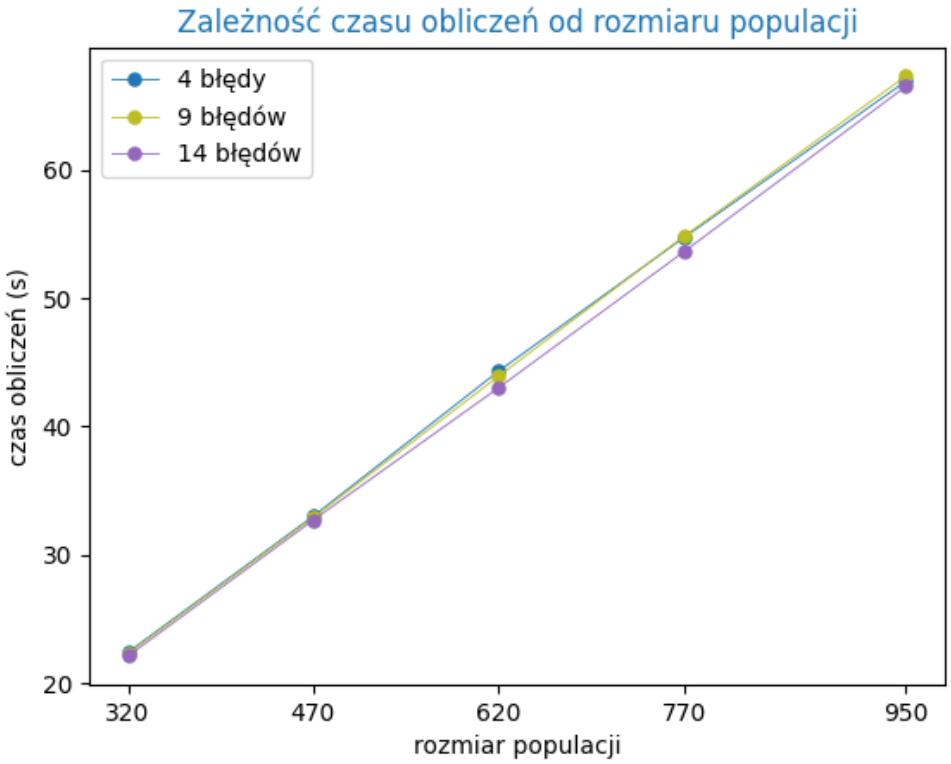
Poziom trudności	4			7			11		
Liczba błędów	3	7	11	3	7	11	3	7	11
f. celu czas	9 47.7s	9.6 47.4s	10.6 45s	15.9 126s	17 125s	17.5 125s	15 99s	16.6 103s	17.8 101s

*podane najlepsze uzyskane wyniki i odpowiadający im czas.

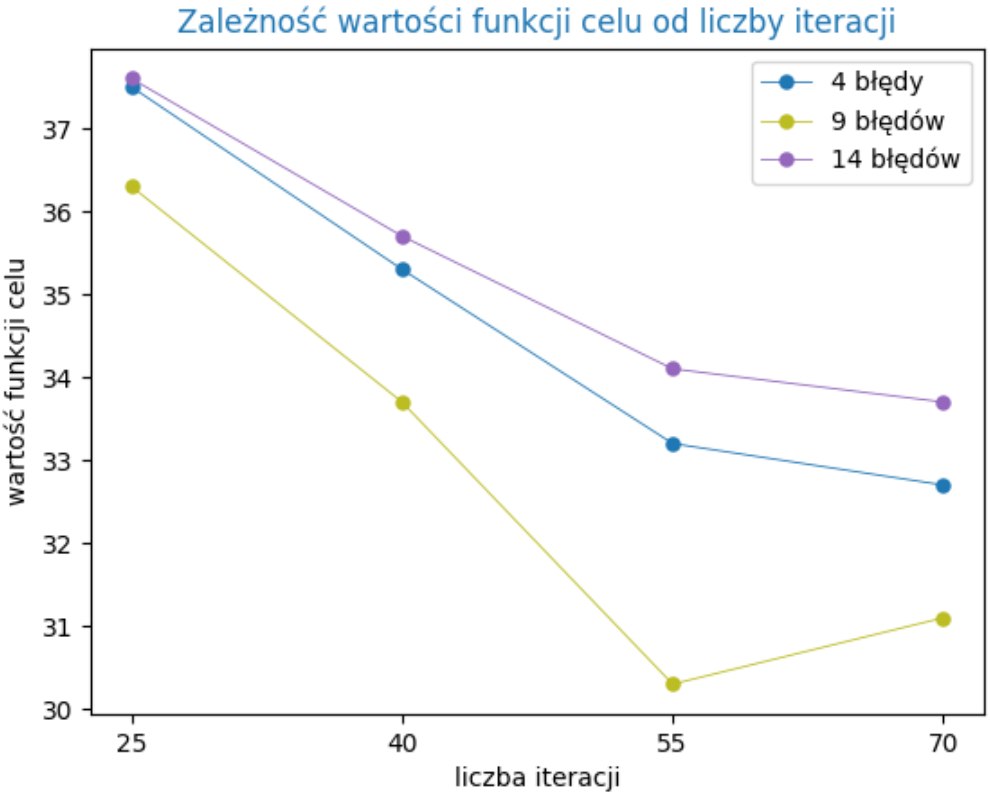
Macierz o wymiarach 20x20 z poziomem trudności 6 oraz 4/9/14 błędami:
Badany parametr: **rozmiar populacji.**
Pozostałe parametry: liczba iteracji – 20, prawdopodobieństwo krzyżowania – 0.5
prawdopodobieństwo mutacji – 0.2

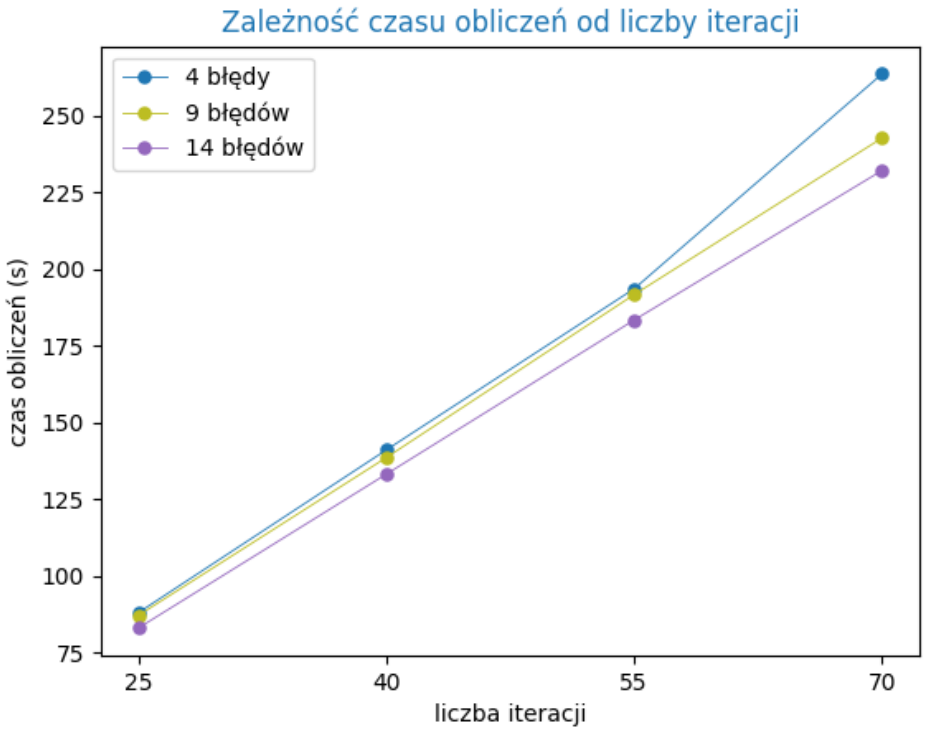


Wnioski:
Ponownie obserwuje się znaczącą poprawę funkcji celu w zależności od parametru rozmiaru populacji, chociaż widać już, że przy tym rozmiarze wyniki te są dość słabe i wartość funkcji celu jest daleka od rozwiązania. Choć i tak dużo lepsza od wartości funkcji celu macierzy przekazywanych na wejściu metaheurystyki (instancji), które z reguły osiągały wartości w przedziale 60-80. Do dalszych testów wybrano rozmiar populacji 950.
Widać także ze czas obliczeń jest już stosunkowo duży przy dość małej liczbie iteracji.



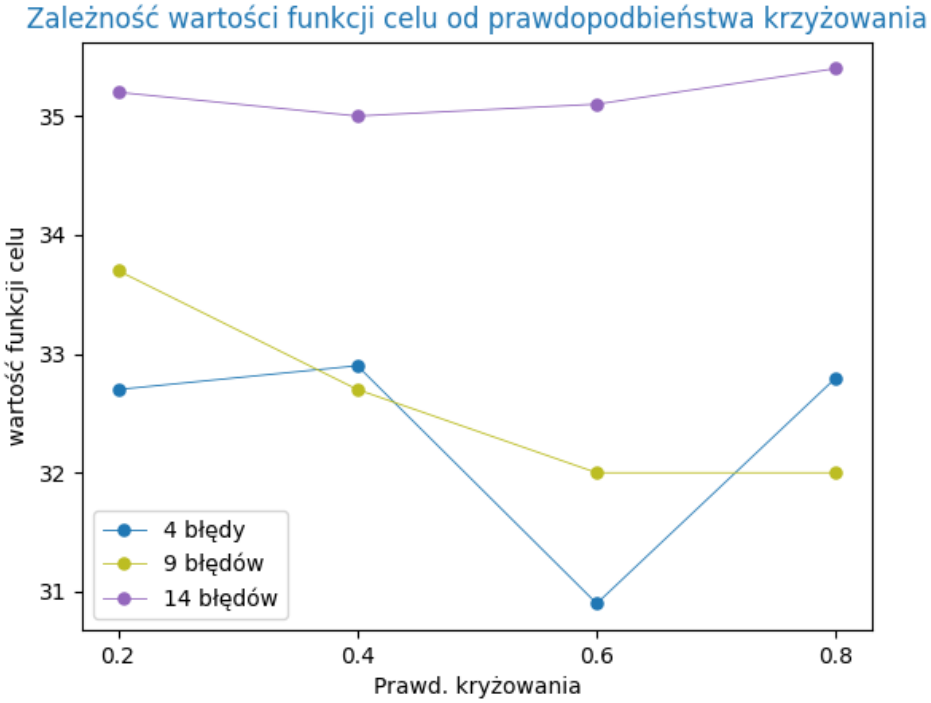
Badany parametr: **liczba iteracji.**
Pozostałe parametry: rozmiar populacji - 950, prawdopodobieństwo krzyżowania – 0.5 prawdopodobieństwo mutacji – 0.2



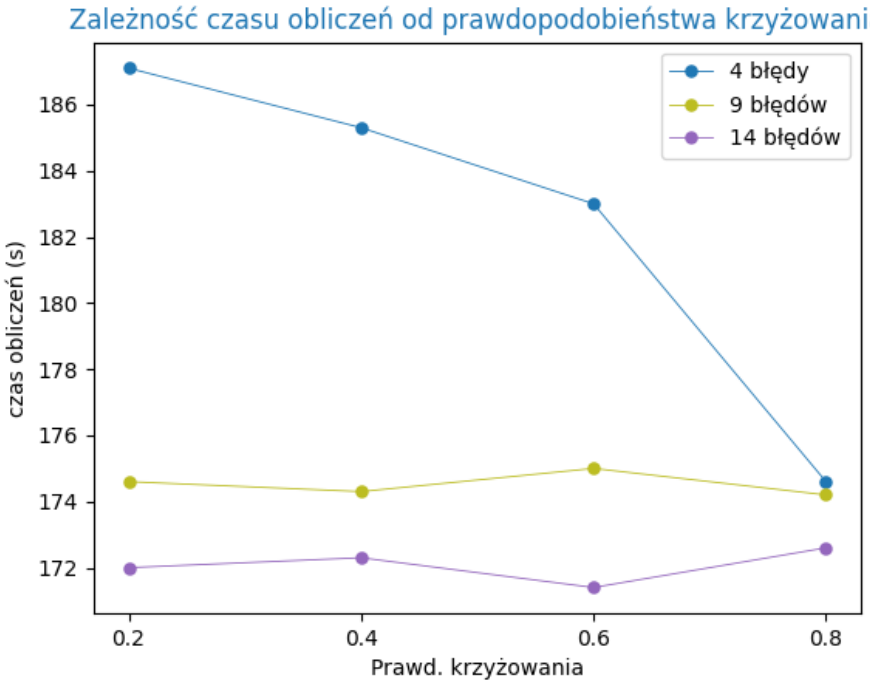


Wnioski:
Ponownie wartość funkcji celu ulega poprawie wraz ze wzrostem liczby iteracji, choć nie są to duże różnice. Do dalszych testów wybrano liczbę 55, gdyż z wykresu wynika że jest to wystarczająca ilość, tylko w przypadku instancji z 4 i 11 błędami lepiej wypadła liczba iteracji 70, lecz jest to delikatna poprawa, a koszt czasowy jest większy jak widać na wykresie powyżej.

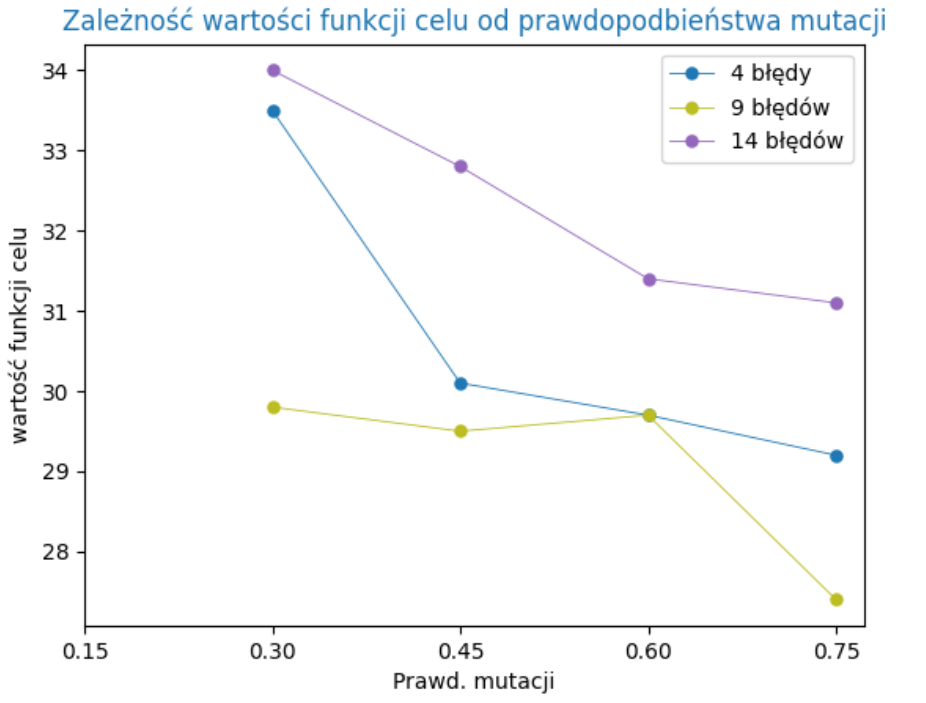
Badany parametr: **prawdopodobieństwo krzyżowania.**
Pozostałe parametry: rozmiar populacji - 950, liczba iteracji – 55 prawdopodobieństwo mutacji – 0.2

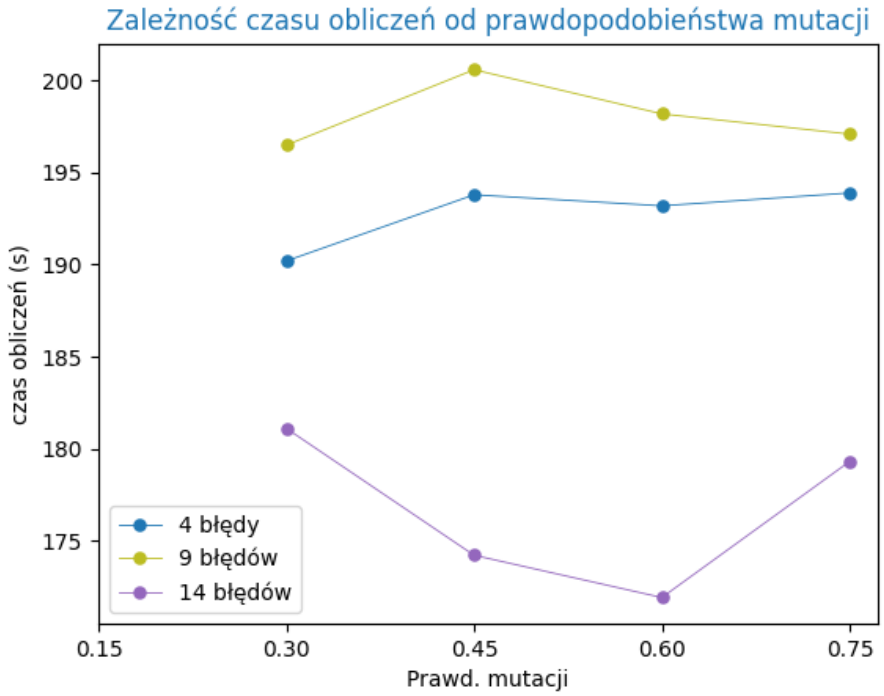


Wnioski:
Bardzo mały wpływ manipulacji tym parametrem na wyniki dla instancji o dużym rozmiarze.
Wybrana wartość prawdopodobieństwa krzyżowania to 0.6
Czas nie jest w ogóle zależny od zmiany tego parametru, choć zaobserwowano dziwny wzrost czasu obliczeń dla instancji z 4 błędami przy niskim prawdopodobieństwie krzyżowania, co jest wręcz odwrotne od spodziewanego zachowania:



Badany parametr: **prawdopodobieństwo mutacji.**
Pozostałe parametry: rozmiar populacji - 950, liczba iteracji – 55 prawdopodobieństwo krzyżowania – 0.6





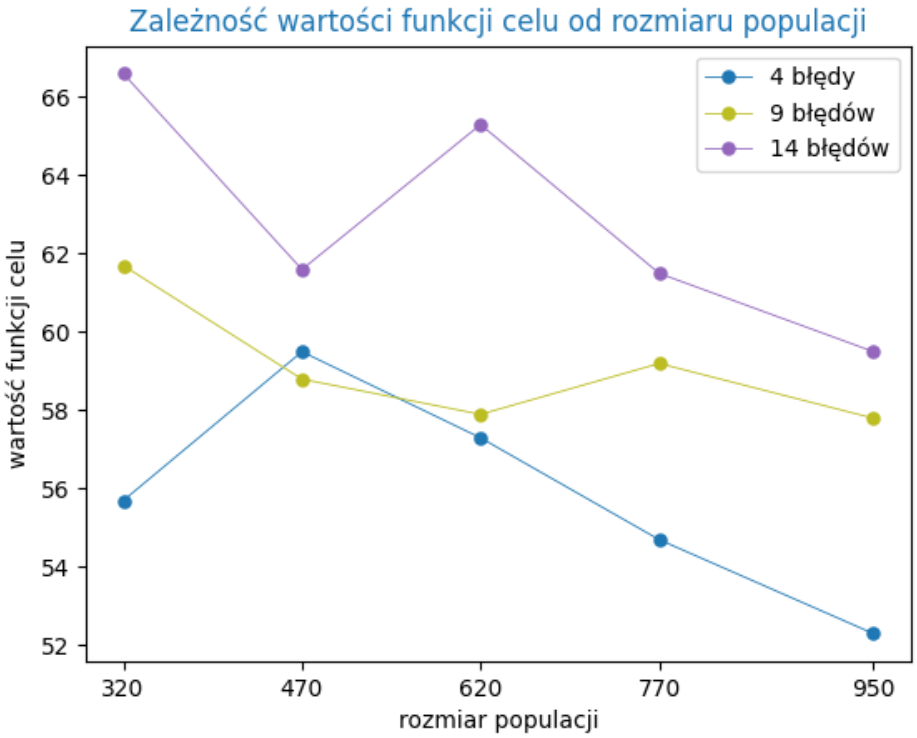
Wnioski:

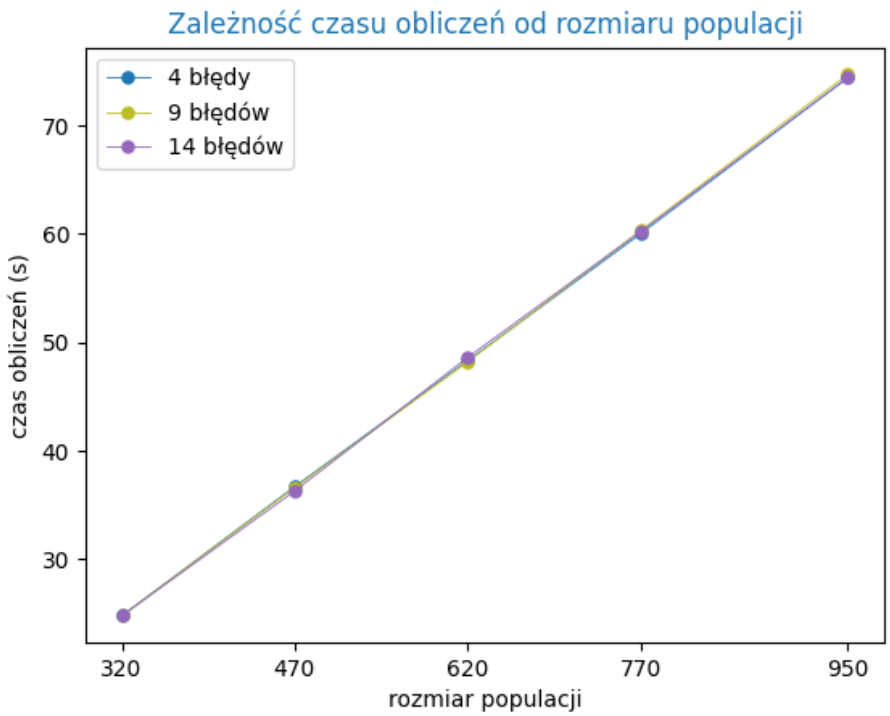
Widać ponownie istotny wpływ zwiększonego prawdopodobieństwa mutacji ma wartość f. celu. Brak natomiast wpływu na czas obliczeń. Różnice są małe i zapewne są kwestią przypadku. Zaobserwowano tylko średni wzrost czasu obliczeń względem poprzedniego wykresu czas (dla prawd. krzyżowania).

Macierz o wymiarach 20x20 z poziomem trudności 12 oraz 4/9/14 błędami:

Badany parametr: **rozmiar populacji.**

Pozostałe parametry: liczba iteracji – 20, prawdopodobieństwo krzyżowania – 0.5 prawdopodobieństwo mutacji – 0.2



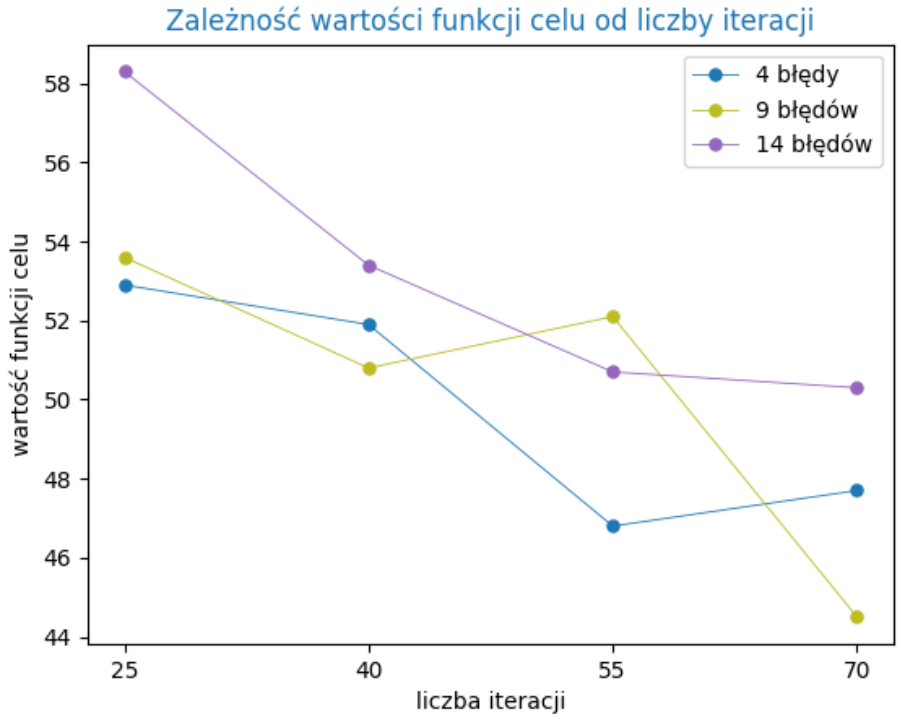


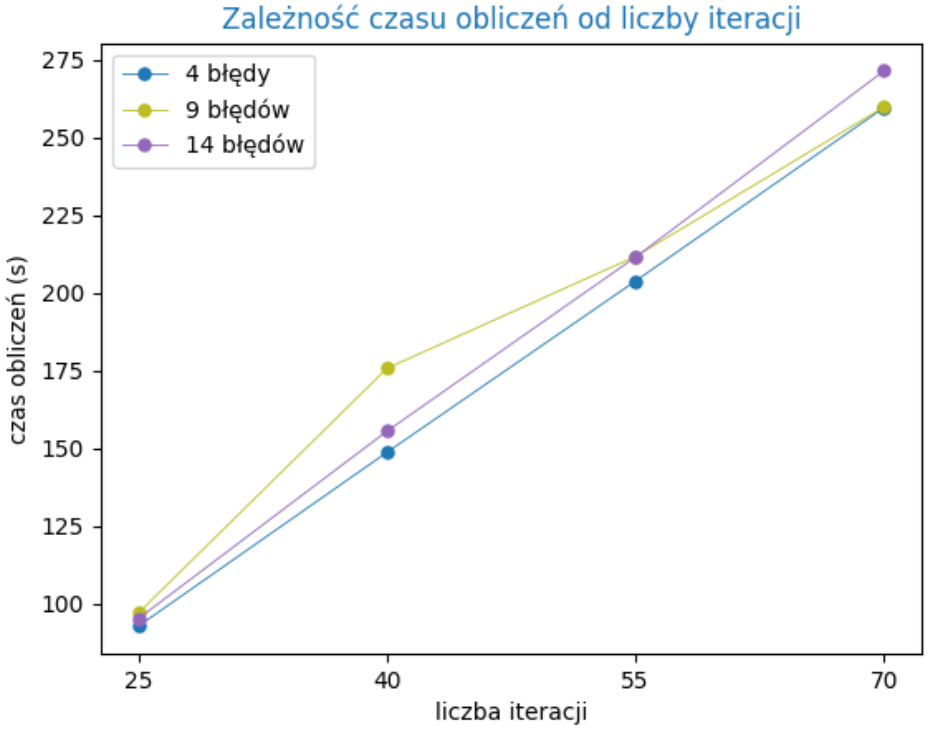
Wnioski:

Poprawa w wartości funkcji celu w gruncie rzeczy nieduża, ale widać, że najlepiej wypada rozmiar populacji 950 i ta wielkość zostaje przyjęta do dalszych testów. Być może korzystnie wpłynie na większą poprawę w zależności od liczby iteracji. Co ciekawe czasy są niemal równe dla instancji z każdą ilością błędów.

Badany parametr: **liczba iteracji.**

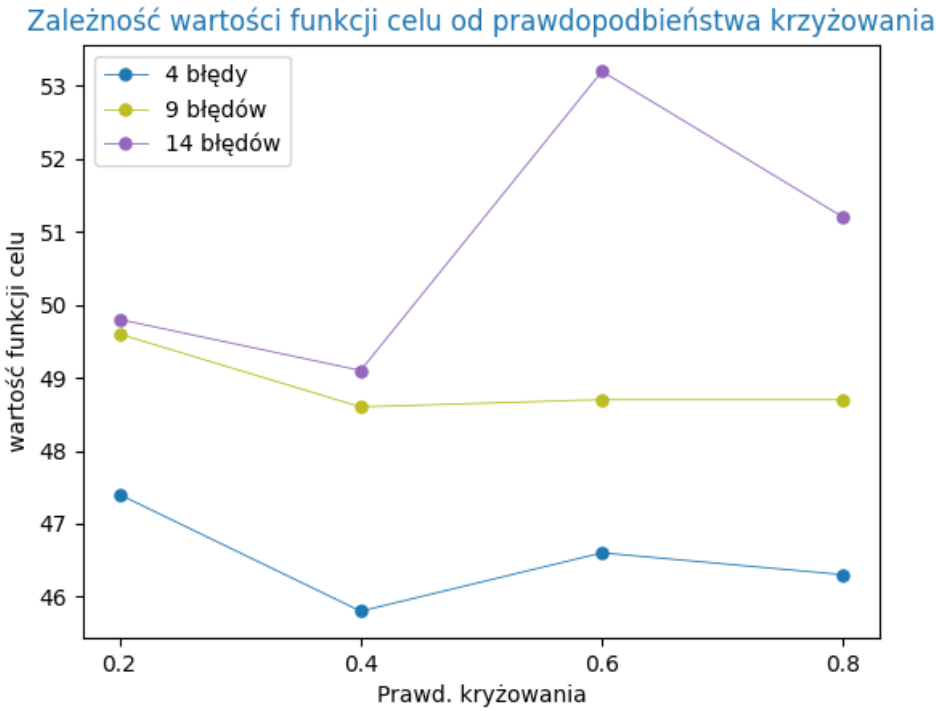
Pozostałe parametry: rozmiar populacji - 950, prawdopodobieństwo krzyżowania – 0.5 prawdopodobieństwo mutacji – 0.2

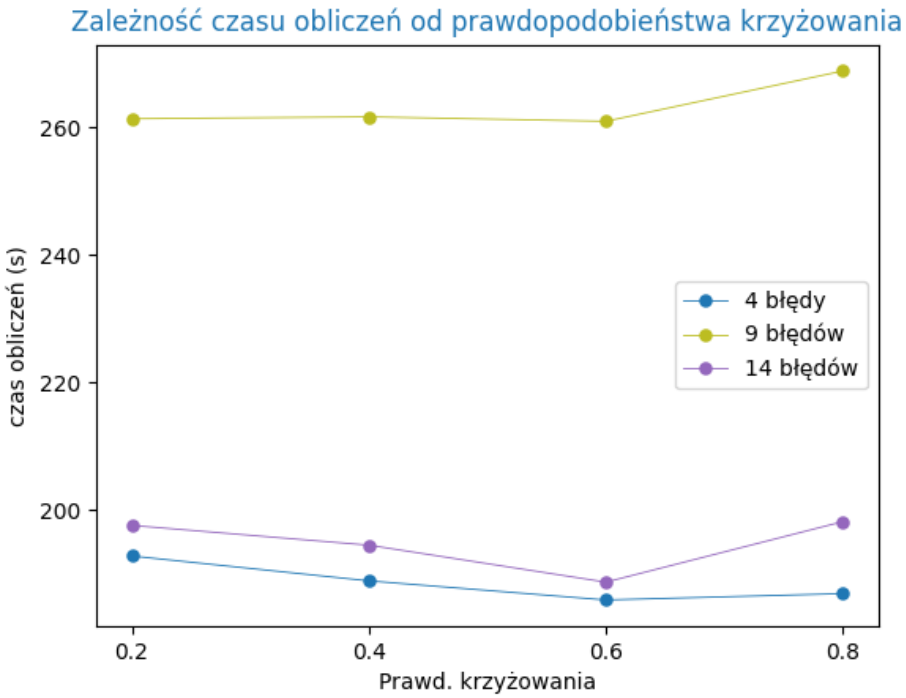




Wnioski:
Również widoczna jest poprawa wraz ze wzrostem liczby iteracji. Wybrano liczbę 55, a dla instancji z 9 błędami liczbę 70 iteracji, ponieważ zaobserwowano dużą różnicę, która raczej nie wynika z losowości.

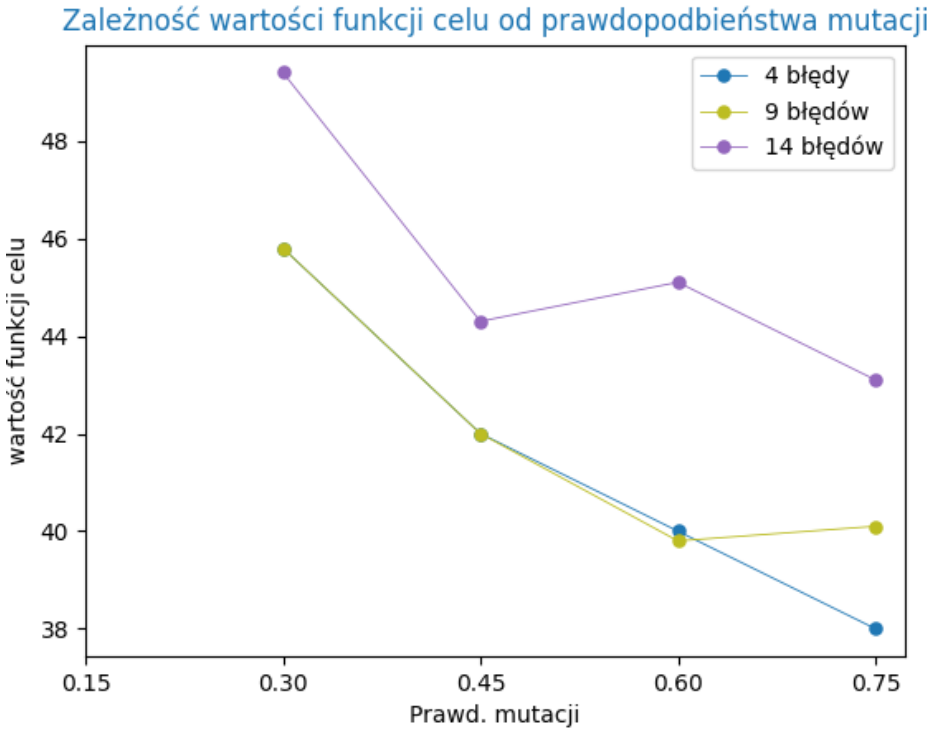
Badany parametr: **prawdopodobieństwo krzyżowania.**
Pozostałe parametry: rozmiar populacji - 950, liczba iteracji – 55/70, prawdopodobieństwo mutacji – 0.2

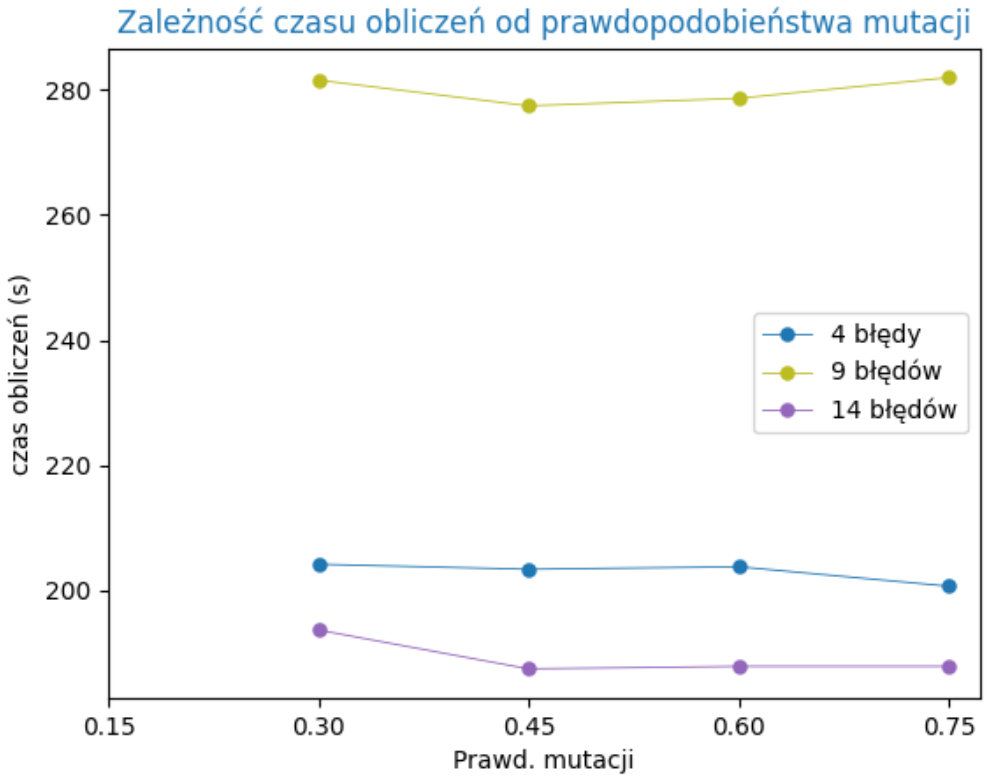




Wnioski:
Dla instancji 20x20 o tym stopniu trudności dla wszystkich stopni wprowadzania błędów najkorzystniej wypada prawdopodobieństwo 0.4, choć zmiany są naprawdę małe.
Czasy osiągają ok 200 sekund a dla instancji z 9 błędami aż 260 sekund. Porównując czasy z poprzedniego wykresu (zależność iteracji) zaobserwować można, że ten nagły wzrost w tych instancjach (9 błędów) jest nienaturalny i nie wiadomo z czego może wynikać.

Badany parametr: **prawdopodobieństwo mutacji.**
Pozostałe parametry: rozmiar populacji - 950, liczba iteracji – 55 prawdopodobieństwo krzyżowania – 0.6





Wnioski:
Ponownie nie wykryto wpływu zmiany tego parametru na czas obliczeń, jedynie wykryto wydłużenie czasu obliczeń dla instancji z 9 błędami względem obliczeń dla testowania prawdopodobieństwa krzyżowania.

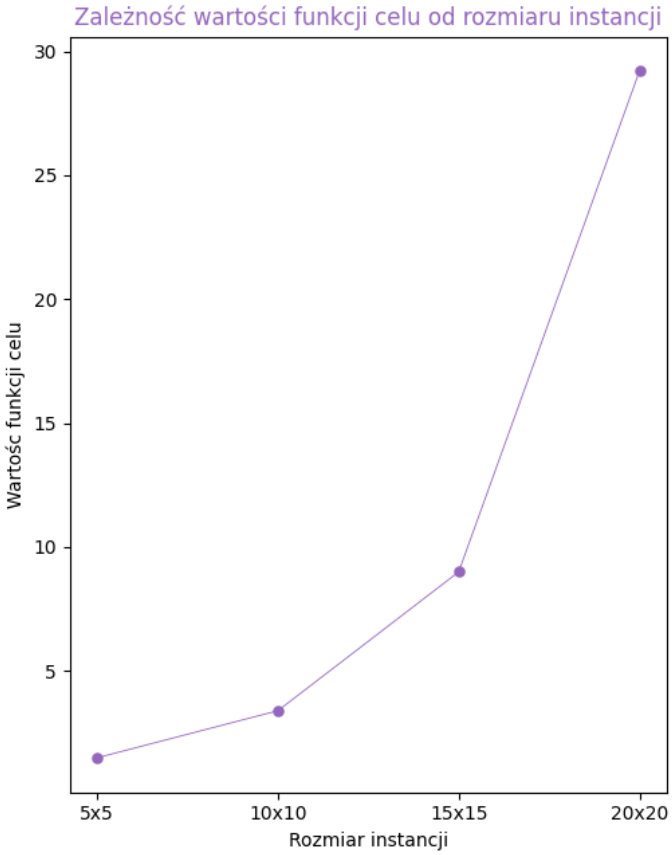
Podsumowanie testowania instancji o rozmiarze 20x20

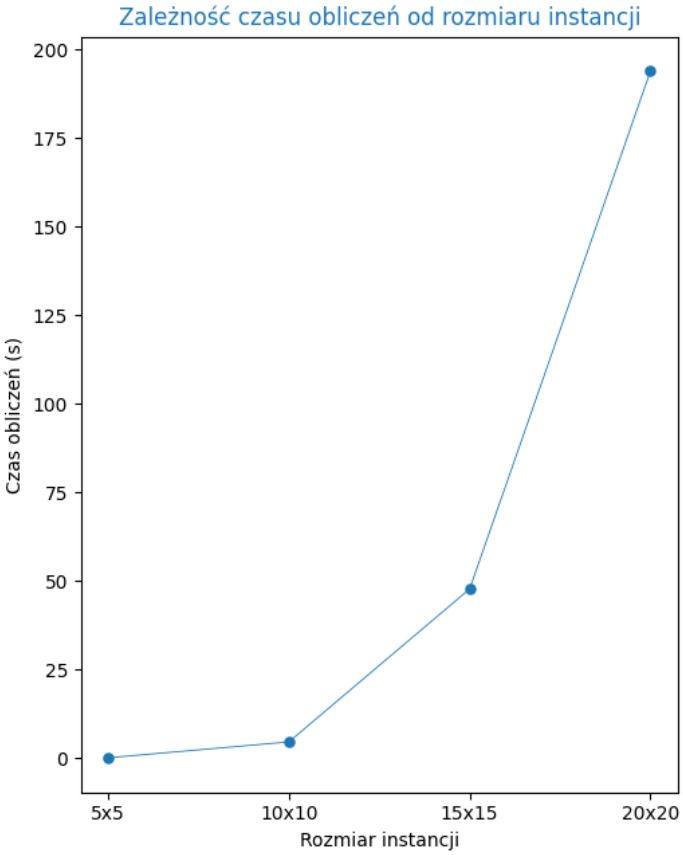
Poziom trudności	6			12		
Liczba błędów	4	9	14	4	9	14
f. celu czas	29.2 193.9s	27.4 197.1s	31.1 179.3s	38 200.7s	39.8 278.6s	43.1 187.9

3.3. Testowanie parametrów instancji

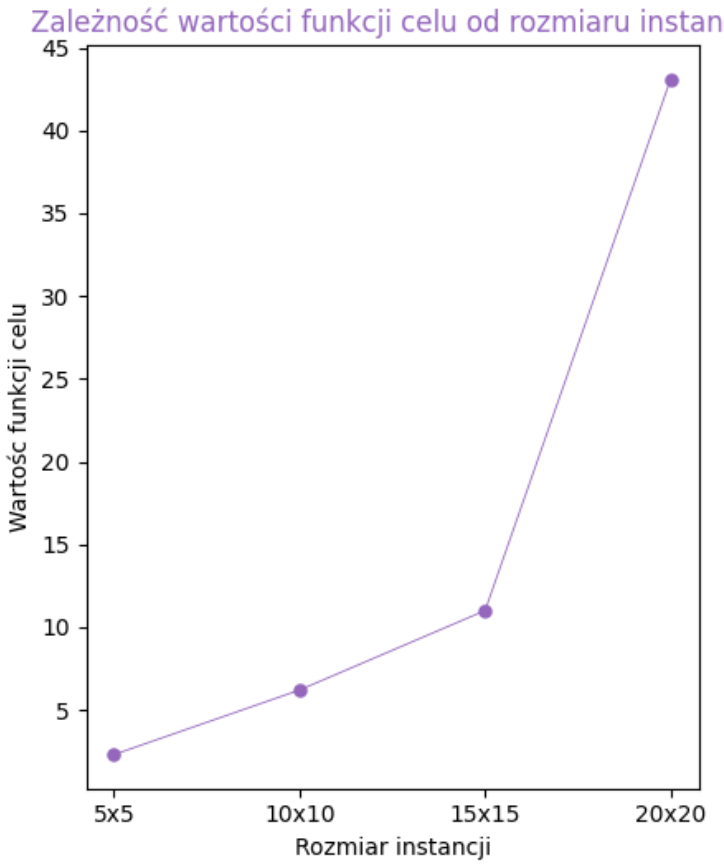
W celu zbadania wpływu rozmiaru instancji stworzono dwa wykresy (co jest bardziej reprezentatywne niż 1 w wypadku zróżnicowanych instancji).
Wzięto dane z dwóch skrajnych kolumn tabel podsumowujących dla każdego rozmiaru instancji, czyli wartości odpowiadających sobie poziomów trudności i odpowiadających sobie ilości wprowadzonych błędów względem rozmiaru instancji.

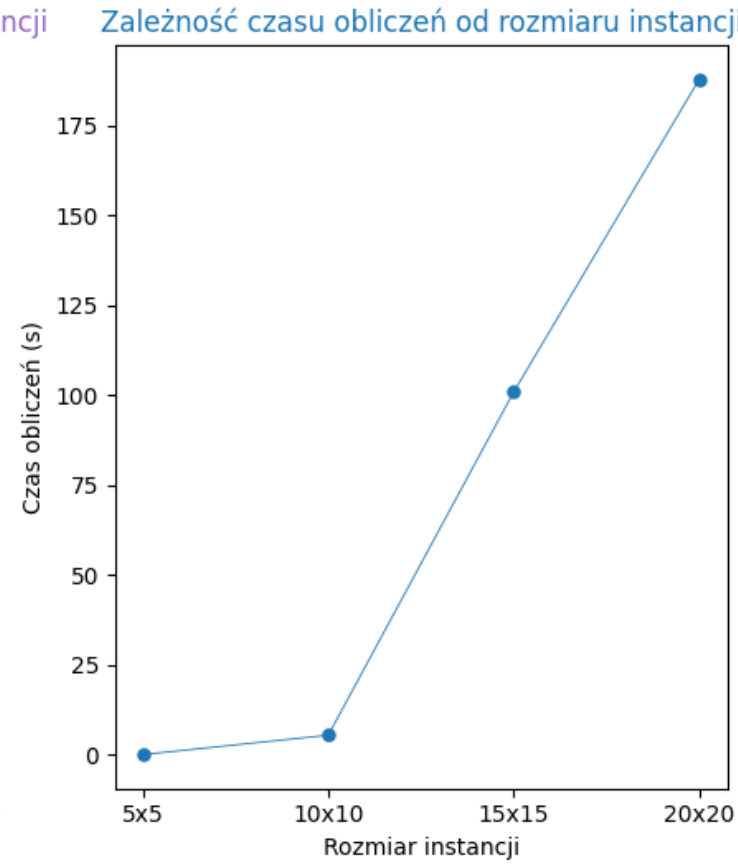
Badany parametr: **rozmiar instancji**
Pozostałe parametry: minimalna ilość błędów dla każdej instancji i minimalny poziom trudności dla każdej instancji





Badany parametr: **rozmiar instancji**
Pozostałe parametry: maksymalna ilość błędów dla każdej instancji i maksymalny poziom trudności dla każdej instancji

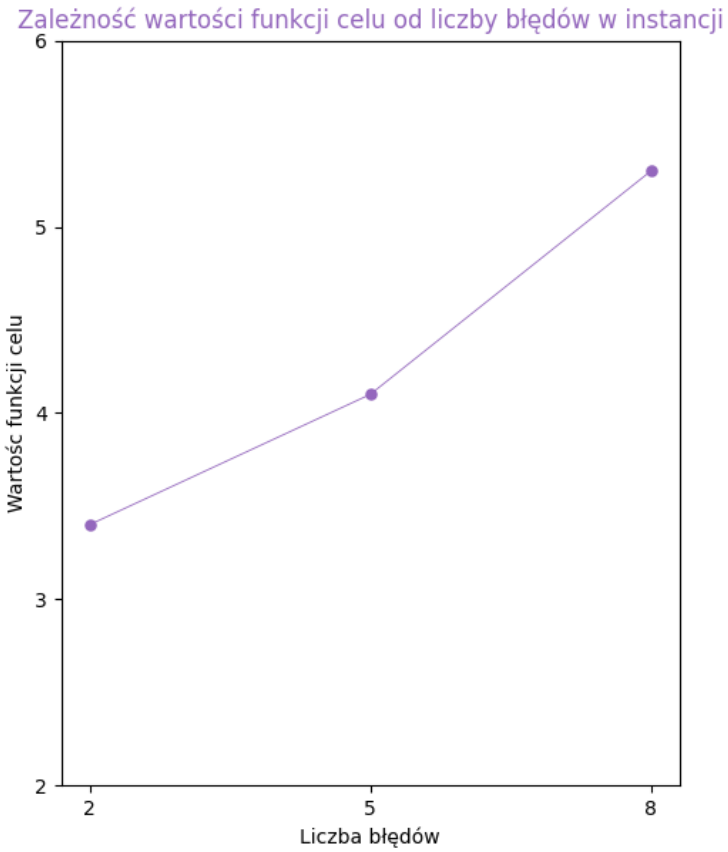


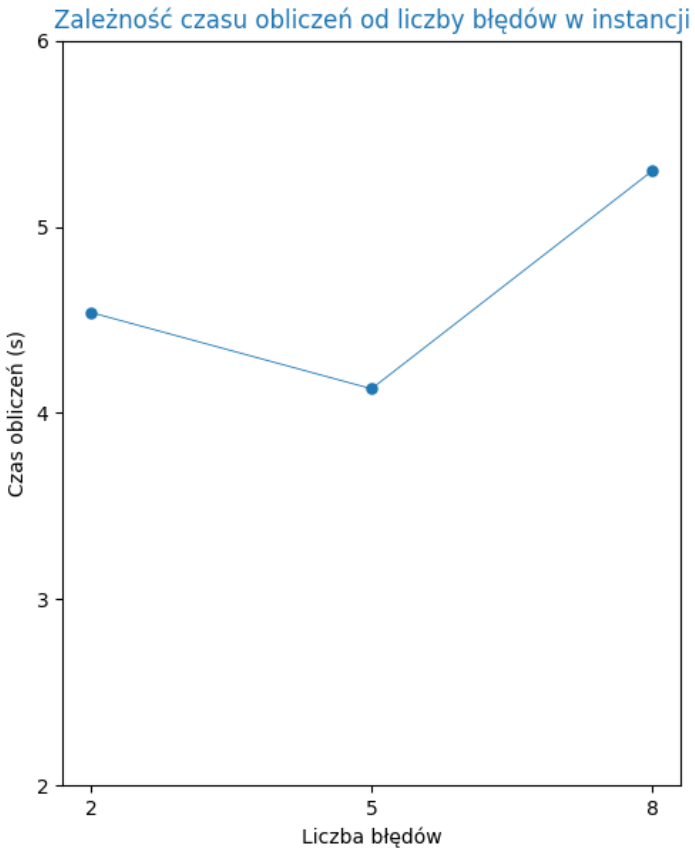


Wnioski:
Widać, że wartość funkcji celu drastycznie rośnie dla rozmiaru większego niż 15x15. Również czas diametralnie się zwiększa, lecz pierwszy duży przeskok czasu obliczeń odbywa się już po przekroczeniu rozmiaru 10x10

Aby zbadać wpływ liczby wprowadzonych błędów sporządzono 3 wykresy. Wzięto macierz o wymiarze 10x10 z najmniejszym stopniem trudności, macierz 15x15 z średnim stopniem trudności oraz macierz 20x20 z największym stopniem trudności.

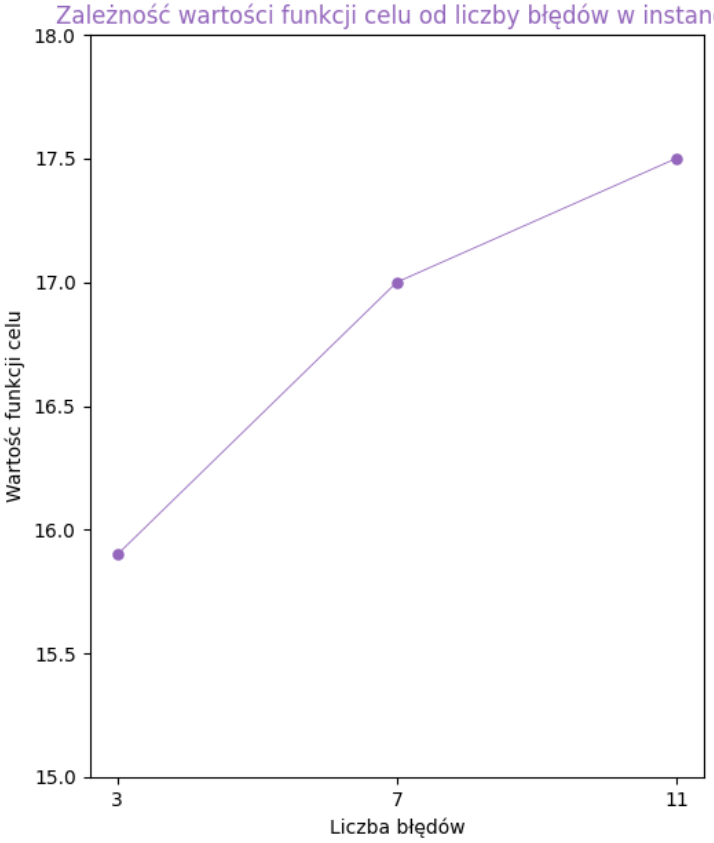
Badany parametr: **liczba błędów w instancji**
Pozostałe parametry: rozmiar 10x10 i najmniejszy stopień trudności.

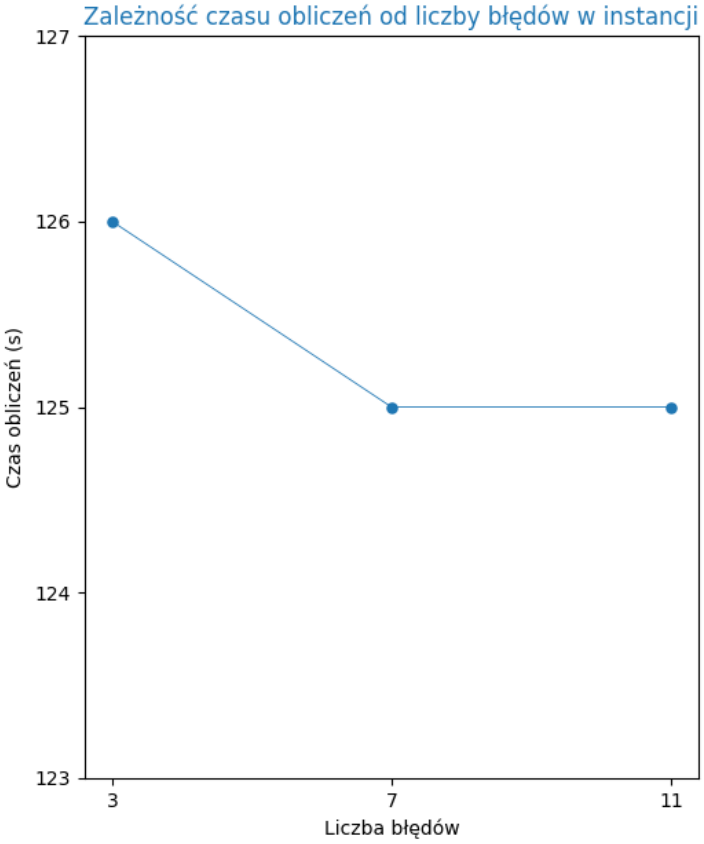




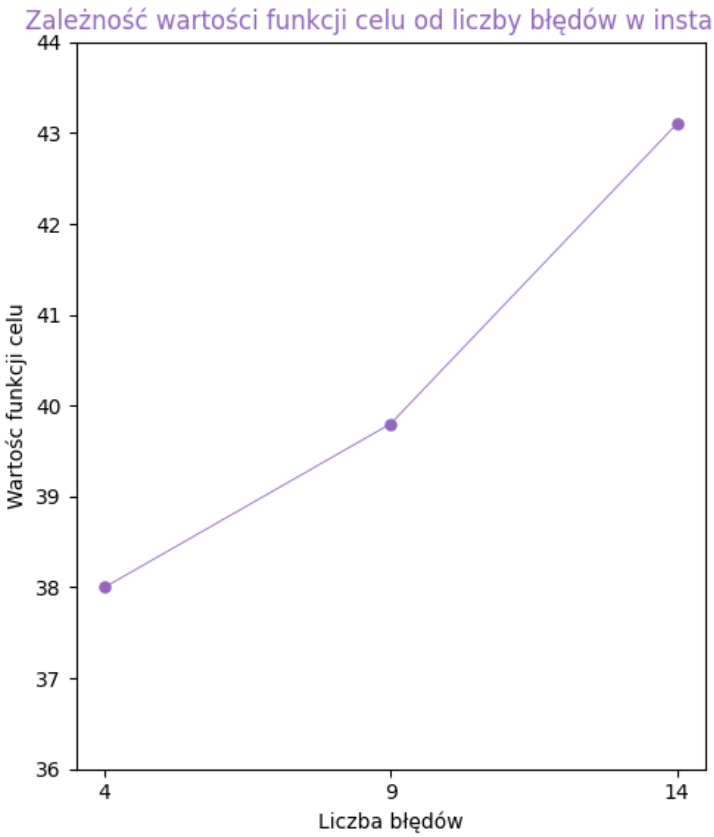
Badany parametr: **liczba błędów w instancji**

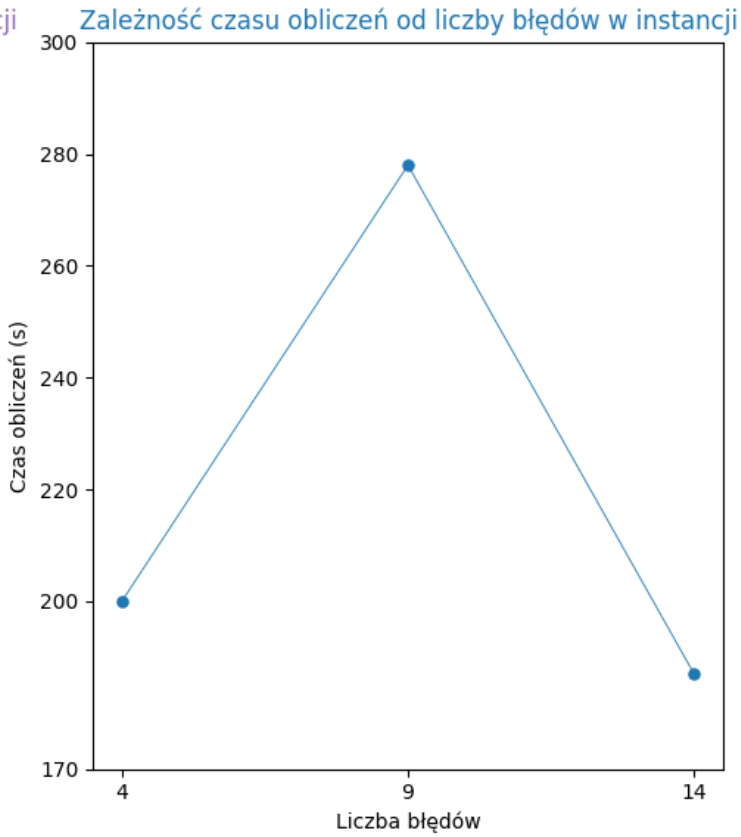
Pozostałe parametry: rozmiar 15x15 i średni stopień trudności.





Badany parametr: **liczba błędów w instancji**
Pozostałe parametry: rozmiar 20x20 i maksymalny stopień trudności.

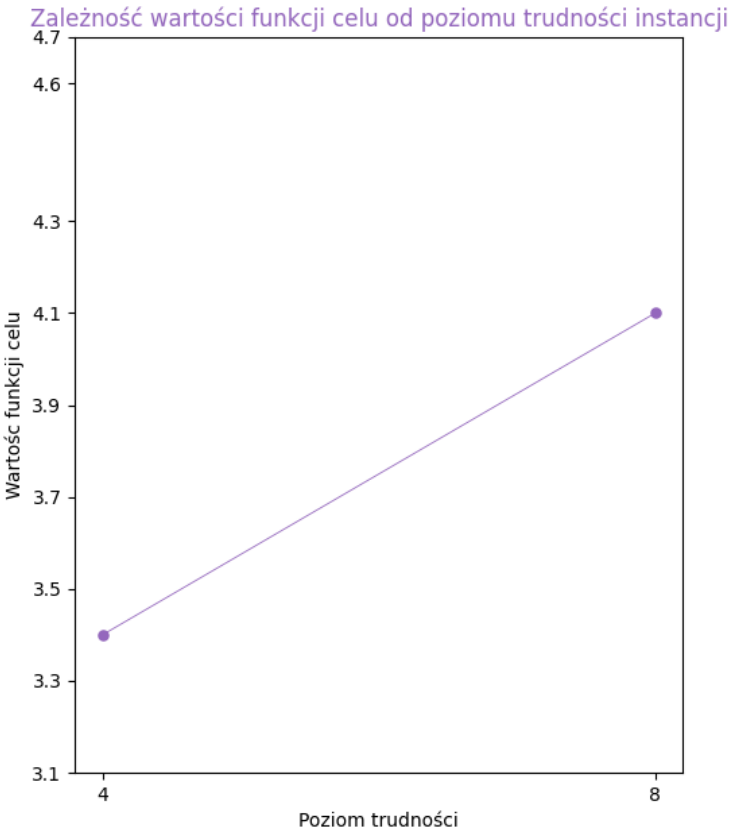


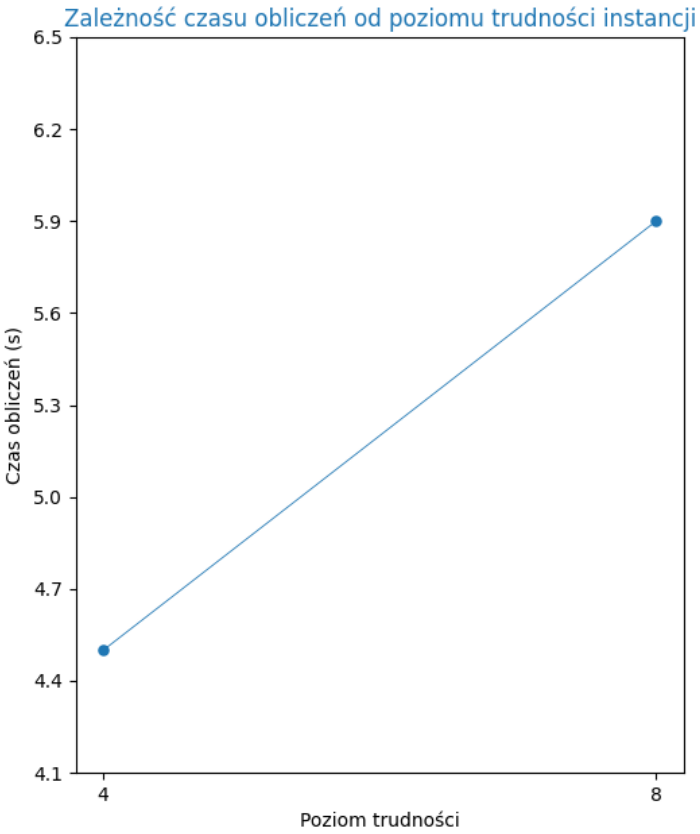


Wnioski:
Zjawisko jakie można zaobserwować to delikatny wzrost funkcji celu wraz ze wzrostem liczby błędów w instancji, lecz wpływ ten jest mały. Generalnie liczba błędów nie wpływa na czas obliczeń. Jedynie na wykresie powyżej widać, że dla liczby błędów 9 czas obliczeń nienaturalnie się wydłuża.

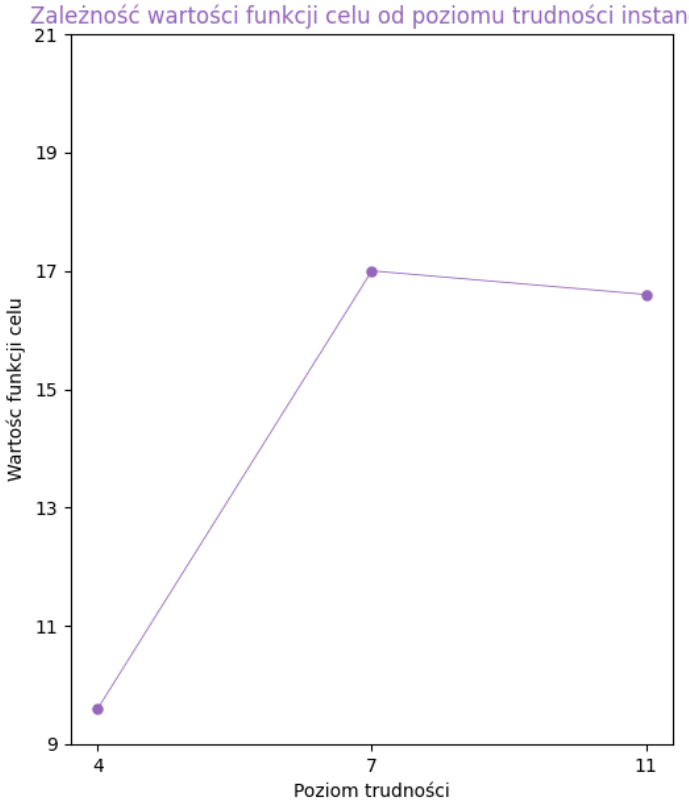
Aby zbadać wpływ poziomu trudności instancji na czas i funkcję celu wzięto dane dla macierzy 10x10 z najmniejszą liczbą błędów, macierzy 15x15 z średnią liczbą błędów oraz macierz 20x20 z największą liczbą błędów.

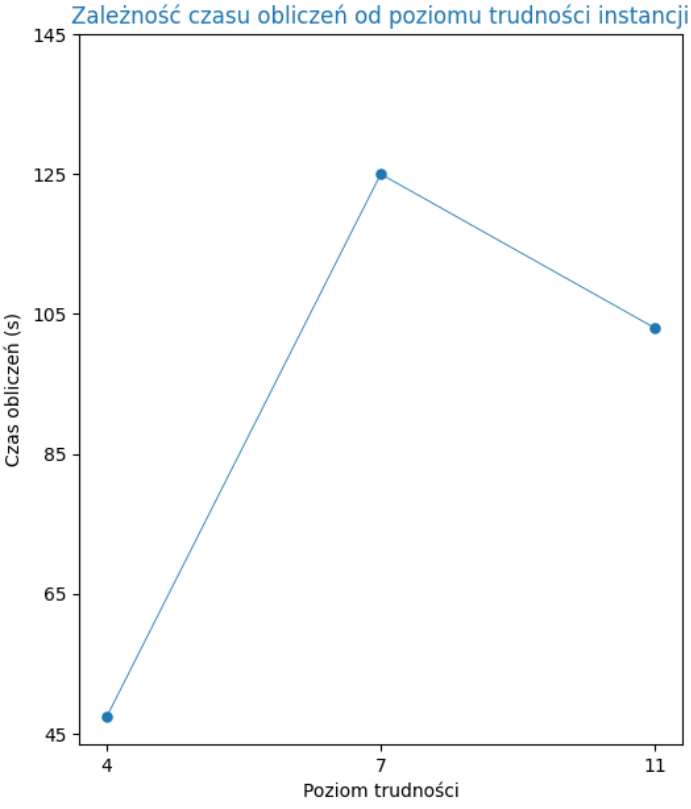
Badany parametr: **poziom trudności instancji**
Pozostałe parametry: rozmiar 10x10 i minimalna liczba błędów.





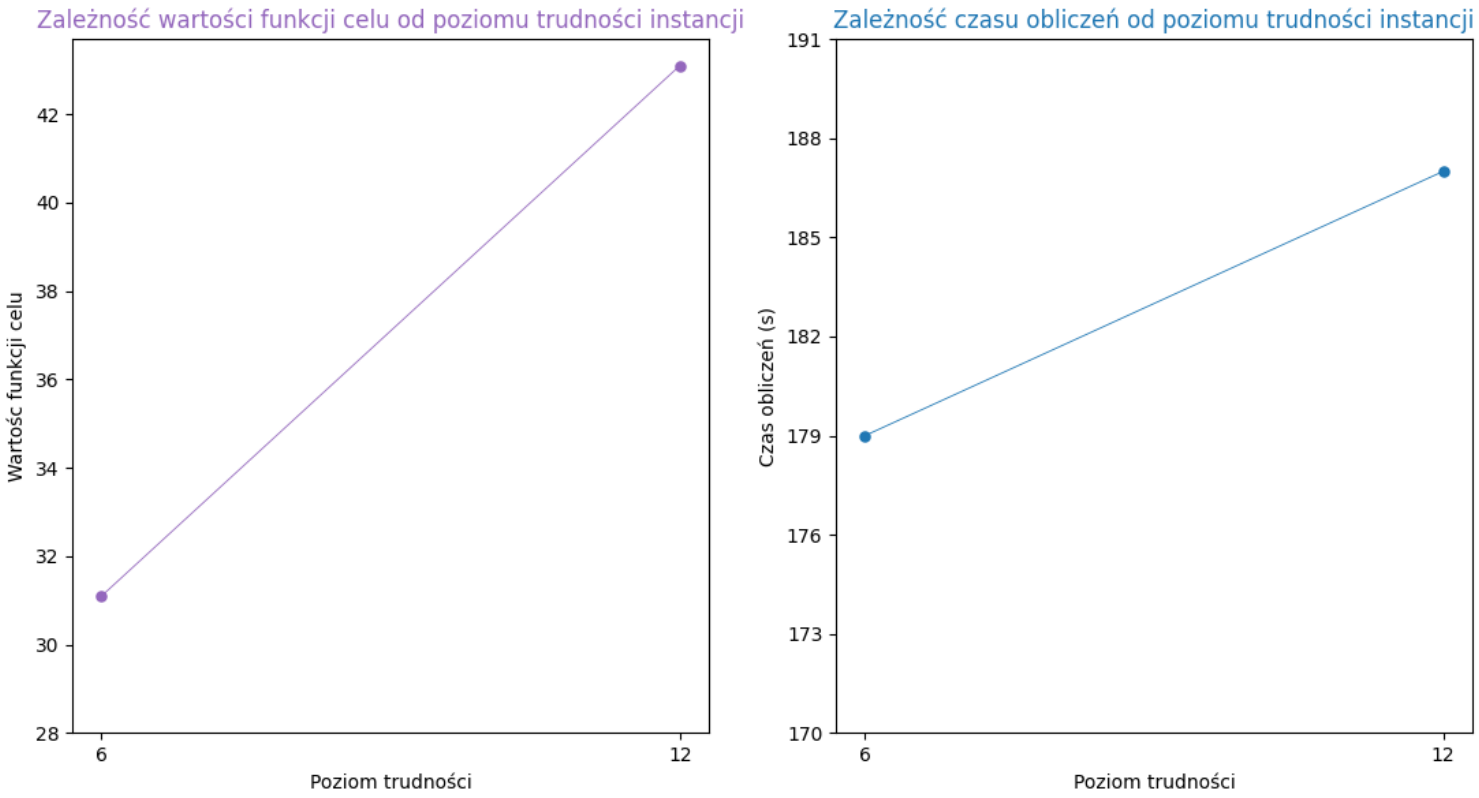
Badany parametr: **poziom trudności instancji**
Pozostałe parametry: rozmiar 15x15 i średnia liczba błędów.





Badany parametr: **poziom trudności instancji**

Pozostałe parametry: rozmiar 20x20 i największa liczba błędów.



Wnioski:

Poziom trudności znacząco wpływa na wartość funkcji celu, ale i także na czas obliczeń. Ciekawym spostrzeżeniem jest, że w przypadku macierzy 15x15 najtrudniejsze instancje są te o średniej wartości poziomu trudności, co w sumie jest dość intuicyjne i burzy koncepcje powiązania nazwy poziomu trudności z rosnącą cyfrą.

4. Wnioski

Zbudowano funkcjonalną i responsywną aplikację, która jest prosta oraz z uwagi na wielowątkowość przyjazna w obsłudze. Umożliwia generowanie instancji oraz rozwiązywanie trudnego obliczeniowo problemu *MIN ERROR MAP* w czasie wielomianowym poprzez zastosowanie algorytmu metaheurystycznego. Zaimplementowany algorytm to algorytm genetyczny, który powinien poprawiać wartość rozwiązania korzystając z uproszczonych praw genetyki panujących w populacjach organizmów.
Przeprowadzono zaawansowane testy i stwierdzono, że zaimplementowany algorytm spełnia swoje zadanie, to jest poprawia wartość funkcji celu w zależności od odpowiednich parametrów i czasu obliczeń, jednak jak zauważono w trakcie testów nie radzi sobie za dobrze z większymi instancjami, czyli takimi które mają rozmiar 20x20 lub większy. Co prawda w miarę działania zbliżają wartość funkcji celu do wartości rozwiązania optymalnego, lecz odległość ta i tak pozostaje spora. Zauważono, że trudność instancji i czas jaki potrzebuje algorytm drastycznie rośnie dla większych instancji. Zwiększanie rozmiaru o 5 z rozmiaru 20x20 do 25x25 drastycznie zwiększa wartości funkcji celu rozwiązań, o wiele bardziej niż zmiana z rozmiaru 10x10 do rozmiaru 15x15.
Z uwagi na sekwencyjne testowanie parametrów metaheurystyki oraz długie czasy obliczeń dla większych instancji testowanie wymagało dużej ilości czasu.
Najwięcej trudności w implementacji algorytmu sprawiło zaprojektowanie funkcji celu oraz napisanie kodu dla niej w taki sposób, aby wszystko działało wedle założeń. Kolejną trudnością było krzyżowanie, ponieważ zauważono, że w trakcie działania programu czasem generowane są rozwiązania niepoprawne, co wymagało czasochłonnej analizy i modyfikacji procedury krzyżującej.
Ostatecznie, mimo że pojedyncze parametry dla większych instancji potrafiły być testowane przez 10 lub więcej godzin, algorytm nigdy nie zgłosił wyjątku ani błędu, co oznacza że cały kod napisany został poprawnie i zgodnie założeniem.
Jedyny problem to niska efektywność metaheurystyki przy instancjach większego rozmiaru, co jest naturalne ponieważ problem jest trudny obliczeniowo. Jednak spodziewano się lepszych wyników dla rozmiaru 20x20, a nawet testowania wymiarów 25x25 i 30x30. Być może po dogłębnej analizie literatury oraz zastosowanych w algorytmie metod, udałoby się poprawić jej działanie.