Projektowanie Efektywnych Algorytmów

Kierunek *Informatyka*	**Termin** *Poniedziałek 15:15*
Temat *Algorytmy populacyjne*	**Problem** *TSP*
Skład grupy *Kinga Marek 235 280*	**Nr grupy** -
Prowadzący *Mgr inż. Radosław Idzikowski*	**data** *January 28, 2020*

1 Opis problemu

Problem komiwojażera (ang. *The travelling Salesman Problem*) to problem optymalizacyjny polegający na znalezieniu w grafie cyklu, który zawiera wszystkie jego wierzchołki (dokładnie jeden raz), a suma wag tego cyklu jest najmniejsza. W bardziej formalnym ujęciu problem ten polega na znalezieniu minimalnego cyklu Hamiltona w grafie.
Nazwa problemu wywodzi się z typowej ilustracji problemu, która przedstawia komiwojażera podróżującego pomiędzy określoną ilością miast w których sprzedaje swoje produkty lub zawiera różne oferty handlowe. Na koniec komiwojażer powraca do swojego miasta rodzinnego. Komiwojażer chce, aby jego trasa była możliwe jak najkrótsza - zagadnienie to rozwiązuje właśnie problem komiwojażera. W rzeczywistości jednak problem ten ma bardzo szerokie zastosowanie w optymalizacji wielu procesów.

2 Algorytmy genetyczny

Algorytm genetyczny to rodzaj heurystyki przeszukującej przestrzeń alternatywnych rozwiązań problemu w celu wyszukiwania najlepszych rozwiązań. Jego twórca John Henry Holland inspirował się biologią, a dokładnie zjawiskiem ewolucji biologicznej.

Podstawowe definicje

Osobnik (*ang. individual*) - pojedyncza propozycja rozwiązania problemu
Populacja - zbiór osobników, na których operuje algorytm
Chromosom - reprezentacja potencjalnego rozwiązania podlegająca ocenie w trakcie działania algorytmu
Genotyp - dziedziczna informacja, w którą wyposażony jest każdy osobnik
Gen - to najmniejszy element zawarty w chromosomie, niosący ze sobą informację genetyczną
Funkcja przystosowania - funkcja zwracająca liczbę, będącą oceną jakości przystosowania osobnika. Przystosowanie osobnika związane jest z jakością danego rozwiązania
Krzyżowanie (*ang. crossing-over*) - proces generowania nowej populacji z osobników, które weszły do grupy rozrodczej
Mutacja - o samoistna, losowa zmiana w genotypie

Przebieg algorytmu

Poniżej w punktach przedstawiono najczęściej stosowane, najbardziej podstawowe działanie algorytmu. Poszczególne implementacje mogą różnić się metodami selekcji, krzyżowania, mutacji i generowania następnej populacji.

1. Losowanie populacji początkowej
2. Populacja poddawana jest ocenie, a następnie najlepiej przystosowane osobniki biorą udział w procesie reprodukcji (czasem ten krok jest pomijany, a osobniki do reprodukcji wybierane są losowo)
3. Tworzenie nowych osobników w procesie krzyżowania
4. W otrzymanej populacji (populacja początkowa i nowe osobniki z punktu 3.) poddawane są mutacji
5. Populacja w obecnym stanie poddawana jest ocenie i zapamiętywane jest najlepsze rozwiązanie (jeśli jest lepsze od obecnego najlepszego znalezionego)
6. Wybór kolejnego pokolenia, czyli następnej populacji
7. Powrót do punktu 2. (dopóki nie stworzono odpowiedniej liczby pokoleń)

3 Implementacja

Parametry algorytmu

Lista parametrów algorytmu genetycznego może różnić się w zależności od szczegółów implementacji i konkretnego problemu, który rozwiązuje. W przygotowanej implementacji stosuje się następujące parametry:

- CrossoverProbability - prawdopodobieństwo dla każdego jednego członka populacji, że zostanie użyty jako rodzic w operacji krzyżowania
- MutationProbability - prawdopodobieństwo dla każdego jednego członka populacji, że zostanie poddany mutacji
- GenerationSize - rozmiar populacji
- MaxNumberOfGenerations - maksymalna liczba generacji tworzonych przez algorytm

Poniżej przedstawiono wywołanie algorytmu z przykładowymi parametrami.

```
func main() {
        path := "C:\\Users\\KM\\Downloads\\PEA\\ATSP\\ATSP\\data100.txt."
        adjacencyMatrix, _ := local.LoadAdjacencyMatrixFromFile(path)
        genetic := genetic.GeneticAlgorithm{}
        genetic.CrossoverProbability = 0.8
        genetic.MutationProbability = 0.03
        genetic.GenerationSize = 100
        genetic.MaxNumberOfGenerations = 2500
        result := genetic.Resolve(adjacencyMatrix)
        cost := local.CalculateCost(result, adjacencyMatrix)
        fmt.Println(cost)
}
```

W listingu powyżej do wywołania algorytmu zostaje użyta funkcja *Resolve*. Funkcja ta inicjuje strukturę wykonującą algorytm, wybiera początkową populację i wywołuje funkcję, która jest odpowiedzialna za stworzenie nowych generacji zadaną ilość razy. W ostatnim kroku funkcja ta zwraca najlepsze znalezione rozwiązanie.

```
func (g GeneticAlgorithm) Resolve(adjacencyMatrix [][]int) []int {
        g.adjacencyMatrix = adjacencyMatrix
        g.size = len(adjacencyMatrix[0])

        initialPopulation := g.generateInitialPopulation()
        g.LoopGenerations(initialPopulation)
        return g.bestPath.path
}
```

Początkowa populacja generowana była losowo za pomocą funkcji przedstawionej poniżej.

```
func (g GeneticAlgorithm) generateInitialPopulation() []Individual {
        initialPopulation :=  make([]Individual, g.GenerationSize)

        for i:=0; i < g.GenerationSize; i++ {
                individual := Individual{}
                initialPopulation[i] = individual.GenerateIndividual(g.size)
        }
        return initialPopulation
}
```

Dla czytelności algorytmu stworzono oddzielną strukturę osobnika - *Individual*, która przechowywała propozycję rozwiązania algorytmu oraz interfejs umożliwiający inicjalizowanie losowo wybranego rozwiązania, przeprowadzanie mutacji i krzyżowania oraz poddawanie ocenie. Poniżej przedstawiono strukturę klasy *Individual* oraz metodę do inicjalizacji losowej rozwiązania *GenerateIndividual*.

```
type Individual struct {
        path []int
        cost int
}

func (ind Individual) GenerateIndividual(size int) Individual {
```

```
        for i:=0; i<size; i++ {
                ind.path = append(ind.path, i)
        }
        for i:=0; i<size; i++ {
                index1 := rand.Intn(size)
                index2 := rand.Intn(size)
                sliceExtensions.SwapOnIndexes(ind.path, index1, index2)
        }

        return ind
}
```

Do operacji mutowania użyto operatora *Swap*, który losował dwa różne indeksy i zamieniał dwa miasta (geny) na tych pozycjach.

```
func (ind Individual) Mutate() Individual {

        index1, index2 := ind.getTwoRandomIndexes(len(ind.path)-1)
        sliceExtensions.SwapOnIndexes(ind.path, index1, index2)

        return ind
}

func (ind Individual) getTwoRandomIndexes(maxIndex int) (index1, index2 int) {
        index1 = rand.Intn(maxIndex)
        index2 = rand.Intn(maxIndex)

        for index1 == index2 {
                index2 = rand.Intn(maxIndex)
        }

        return index1, index2
}
```

Do operacji krzyżowania zaimplementowano krzyżowanie dwupunktowe OX - Order Crossover Operator. Jego implementację przedstawiono poniżej.

```
func (ind Individual) Crossover(individual Individual) (child1, child2 Individual) {

        p1 := make([]int, len(ind.path))
        copy(p1, ind.path)
        p2 := make([]int, len(individual.path))
        copy(p2, individual.path)
        o1 := make([]int, len(p1))
        o2 := make([]int, len(p1))

        index1, index2 := ind.getTwoRandomIndexes(len(p1)-1)
        if index1 > index2 {
                replaced := index1
                index1 = index2
                index2 = replaced
        }

        map1 := make(map[int]bool, len(p1))
        for i:=0; i<len(p1); i++ {
                map1[i] = false
        }
        for i:=index1; i<index2; i++ {
                map1[p1[i]]=true
```

```
}

map2 := make(map[int]bool, len(p1))
for i:=0; i<len(p1); i++ {
        map2[i] = false
}
for i:=index1; i<index2; i++ {
        map2[p2[i]]=true
}

copy(o1[index1:index2], p1[index1:index2])
copy(o2[index1:index2], p2[index1:index2])

// from index2 to the end
otherParent := index2
for i:= index2; i<len(p1); i++ {
        indexAssigned := false
        for !indexAssigned {
                if !map1[p2[otherParent]] {
                        map1[p2[otherParent]] = true
                        o1[i] = p2[otherParent]
                        indexAssigned = true
                }
                otherParent++
                if otherParent == len(p1) {
                        otherParent = 0
                }
        }
}

// from start to index1
for i:= 0; index1>i; i++ {
        indexAssigned := false
        for !indexAssigned {
                if !map1[p2[otherParent]] {
                        map1[p2[otherParent]] = true
                        o1[i] = p2[otherParent]
                        indexAssigned = true
                }
                otherParent++
                if otherParent == len(p1) {
                        otherParent = 0
                }
        }
}

// from index2 to the end
otherParent = index2
for i:= index2; i<len(p1); i++ {
        indexAssigned := false
        for !indexAssigned {
                if !map2[p1[otherParent]] {
                        map2[p1[otherParent]] = true
                        o2[i] = p1[otherParent]
                        indexAssigned = true
                }
                otherParent++
                if otherParent == len(p1) {
```

```
                        otherParent = 0
                }
        }
}

// from start to index1
for i:= 0; index1>i; i++ {
        indexAssigned := false
        for !indexAssigned {
                if !map2[p1[otherParent]] {
                        map2[p1[otherParent]] = true
                        o2[i] = p1[otherParent]
                        indexAssigned = true
                }
                otherParent++
                if otherParent == len(p1) {
                        otherParent = 0
                }
        }
}

child1 = Individual{
        path: o1,
        cost: 0,
}
child2 = Individual{
        path: o2,
        cost: 0,
}
return child1, child2
}
```

Powyżej wymienione operatory krzyżowania i mutacji wykorzystywane były w funkcji *LoopGerations* wywoływanej przez funkcję *Resolve* po ustaleniu generacji początkowej. Jej implementację przedstawiono poniżej.

```
func (g *GeneticAlgorithm) LoopGenerations(initialPopulation []Individual) {
        var parentIndex int
        var children []Individual
        currentPopulation := initialPopulation

        for generationNumber := 0; generationNumber < g.MaxNumberOfGenerations;
            generationNumber++ {
                // CROSSOVER
                for index:=0; index < g.GenerationSize; index++ {
                        shouldCrossOver := rand.Float64()

                        if shouldCrossOver > g.CrossoverProbability {
                                parentIndex = rand.Intn(g.GenerationSize-1)
                                for index == parentIndex {
                                        parentIndex =rand.Intn(g.GenerationSize-1)
                                }
                                child1, child2 := currentPopulation[index]
                                    .Crossover(currentPopulation[parentIndex])
                                if len(child1.path) > 0 {
                                        children = append(children, child1)
                                        children = append(children, child2)
                                }
                        }
```

```
                }
                // MUTATE
                for index:=0; index < g.GenerationSize; index++ {
                        shouldMutate := rand.Float64()

                        if shouldMutate < g.MutationProbability {
                                currentPopulation[index] = currentPopulation[index]
                                    .Mutate()
                        }
                }

                if generationNumber == int(float64(g.MaxNumberOfGenerations)*0.8) {
                        g.MutationProbability = g.MutationProbability*2
                }
                currentPopulation = g.nextPopulation(currentPopulation, children)
                children = []Individual{}
        }
}
```

Funckja *LoopGerations* po deklaracji i/lub inicjalizacji zmiennych pomocniczych generuje kolejne operacje w pętli $MaxNumberOfGenerations$ razy. W każdej pętli najpierw wykonywane jest krzyżowanie - każdy osobnik populacji staje się rodzicem dla kolejnego potomka z prawdopodobieństwem $CrossoverProbability$. Wszyscy stworzeni potomkowe zapmiętywani są na liście *children*. W kolejnym etapie każdy osobnik populacji poddawany jest mutacji z prawdopodobieństwem $MutationProbablity$. Każdą iteracji pętli kończy wywołanie funkcji pomocniczej *nextPopulation*, która na podstawie przekazanej listy potomków *children* i obecnej populacji *currentPopulation* tworzy kolejną generacje. Ostatnim krokiem pętli jest usunięcie wszystkich potomków z listy. Implementację metody *nextPopulation* przedstawiono poniżej.

```
func (g *GeneticAlgorithm) nextPopulation
    (currentPopulation []Individual, children []Individual) []Individual{
        var nextPopulation []Individual

        nextPopulation = append(nextPopulation, children...)
        g.calculateWeights(currentPopulation)
        sort.Slice(currentPopulation, func(i, j int) bool {
                return currentPopulation[i].cost < currentPopulation[j].cost
        })

        individualsThatSurviveCount := g.GenerationSize - len(children)
        if len(g.bestPath.path) == 0 {
                g.bestPath= currentPopulation[0]
        } else {
                if g.bestPath.CalculateCost(g.adjacencyMatrix) > currentPopulation[0].cost {
                        g.bestPath = currentPopulation[0]
                }
        }

        nextPopulation = append
            (nextPopulation, currentPopulation[0:individualsThatSurviveCount]...)
        return nextPopulation
}

func (g *GeneticAlgorithm) calculateWeights(population []Individual) {
        for index := 0; index < len(population); index++ {
                population[index].CalculateCost(g.adjacencyMatrix)
        }
}
```

Algorytm tworzenia nowej generacji zaczyna działanie od wywołania pomocniczej metody

calculateWeights, która wywołuje funkcję przystosowania *CalculateCost* w każdym osobniku. Po ocenie każdego osobnika poprzedniej populacji do listy osobników nowej generacji dodawani są wszyscy potomkowie stworzeni podczas krzyżowania. Następnie obliczana jest rożnica między rozmiarem populacji, a ilością wytworzonych potomków - taką ilością osobników z populacji rodziców zostanie dopełniona lista osobników nowej generacji. Osobnicy nowej generacji są sortowani na podstawie ustalonego kosztu przez funkcję przystosowania i najlepsze osobniki są używane do dopełnienia listy osobników nowej generacji.

4 Eksperymenty obliczeniowe

Obliczenia zastały wykonane na komputerze klasy PC z procesorem i5-5200U, kartą graficzną NVIDIA GeForce GTX 940M, 12GB RAM i DYSK SSD. Jako miarę jakości algorytmu przyjęto uśrednione wyniki (pomiary powtórzono 100-krotnie).

Liczba generacji	1000		1500		2000		2500		3000	
Rozmiar	Wynik	Błąd	Wynik	Błąd	Rezultat	Błąd	Wynik	Błąd	Wynik	Błąd
48	16243.9	0.126	16046.3	0.113	16155.6	0.120	15866.7	0.100	15940.3	0.105
53	8447.4	0.223	8367.2	0.212	8303.6	0.203	8509.4	0.232	8317.2	0.205
70	43062.8	0.114	42653	0.103	42956.3	0.111	42641.2	0.103	42055.2	0.087
100	57374.4	0.584	50157.4	0.384	48804.9	0.347	49481.3	0.366	47537.8	0.312
323	3167.6	1.389	2896.3	1.184	2668.3	1.012	2468.2	0.861	2291.8	0.728
403	4515.8	0.832	4323.4	0.754	4092.6	0.660	3988.6	0.618	3853.8	0.563
443	5063.6	0.862	4790.5	0.761	4528.2	0.665	4422.4	0.626	4279.7	0.573

Table 1: Wyniki dla asymetrycznych danych o zadanej liczbie generacji dla prawdopodobieństwa mutacji 3% i prawdopodobieństwa krzyżowania 80%, rozmiaru populacji 100. Przedstawiony błąd to błąd względny.

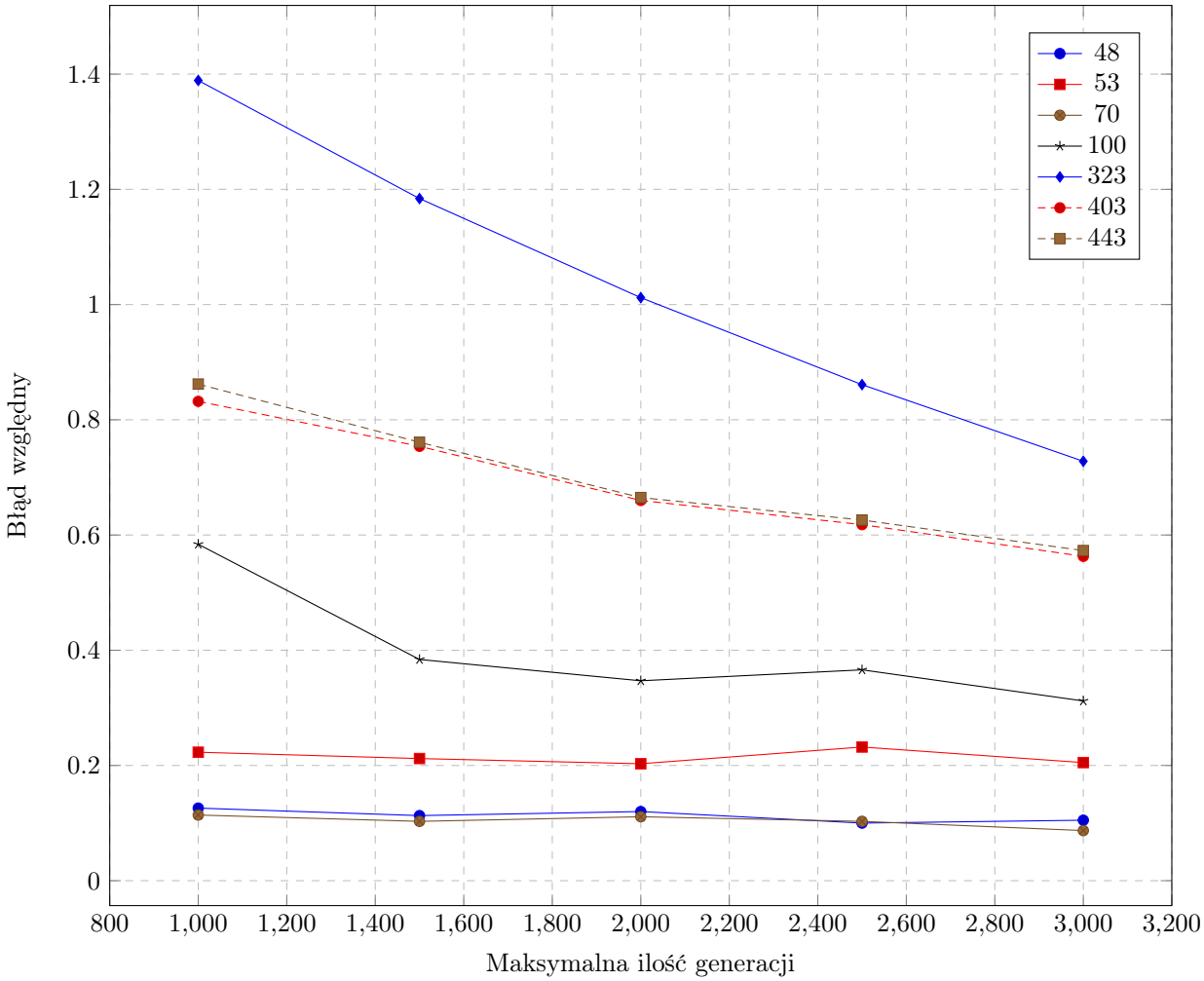

Figure 1: Błąd względny w zależności od maksymalnej ilości generacji dla prawdopodobieństwa mutacji 3% i prawdopodobieństwa krzyżowania 80%, rozmiaru populacji 100. Przedstawiony błąd to błąd względny.

Na wykresie powyżej widać, że dla małych instancji problemu (rozmiar 48, 53) wystarcza niska liczba generacji i dalsze zwiększanie ich liczby nie przenosi żadnych korzyści. Dla większych instancji efekty są różne i charakterystyczne dla konkretnej instancji problemu. Dla instancji o rozmiarze 323 można zobaczyć bardzo silnie i prawie doskonale liniowe zmniejszenie się błędu względnego. Dla rozmiaru 443 możemy także zaobserwować zmniejszenie się błędu względnego wraz ze wzrostem liczby generacji, lecz nie jest ono aż tak duże jak dla rozmiaru 323. Szybkość zmiany jakości wyników w zależności od ilości iteracji dla większych instancji problemów nie zależy wprost od rozmiaru problemu.

Wielkość populacji	20		50		100		150		200		300	
Rozmiar	Wynik	Błąd	Wynik	Błąd	Rezultat	Błąd	Wynik	Błąd	Wynik	Błąd	Rozmiar	Błąd
48	19297.6	0.338	16710.6	0.159	16112	0.117	15656.5	0.086	15286	0.060	15002.1	0.040
53	10112.4	0.465	9350.8	0.354	8409.2	0.218	8107.3	0.174	7913.6	0.146	7646.9	0.107
70	47073.8	0.217	43712.2	0.130	42114.4	0.089	41449	0.072	41366.2	0.070	40639.7	0.051
100	69141.9	0.908	54020	0.491	47495.1	0.311	45331.4	0.251	43190.7	0.192	41353	0.141
323	2996.1	1.260	2534.3	0.911	2431.9	0.834	2469.8	0.863	2433	0.835	2575.3	0.942
403	4258.1	0.727	3995.3	0.621	3973	0.612	3987	0.617	3953.7	0.604	4018.2	0.630
443	4732.8	0.740	4430.5	0.629	4436.6	0.631	4458.2	0.639	4481.2	0.648	4439.3	0.632

Table 2: Wyniki dla asymetrycznych danych o zadanej wielkości populacji dla prawdopodobieństwa mutacji 3% i prawdopodobieństwa krzyżowania 80%, maksymalnej ilości generacji 2500. Przedstawiony błąd to błąd względny.

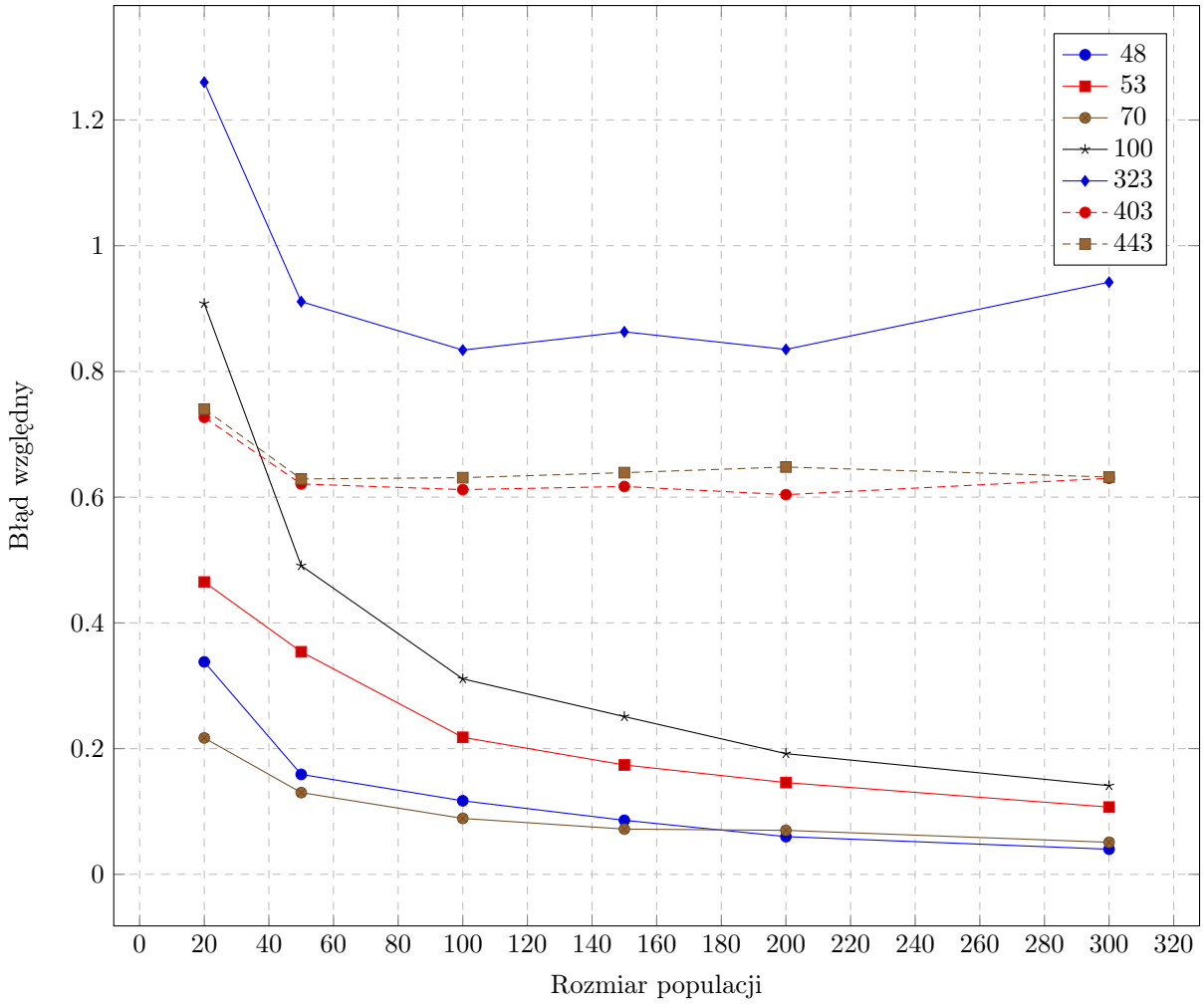

Figure 2: Błąd względny w zależności od rozmiaru populacji dla prawdopodobieństwa mutacji 3% i prawdopodobieństwa krzyżowania 80%, maksymalnej ilości generacji 2500. Przedstawiony błąd to błąd względny.

Na powyższym wykresie widać, że rozmiar populacji poniżej 100 znacznie zwiększa błąd względny dla większości instancji problemu. Dalsze powiększanie wielkości rozmiaru populacji powyżej 150 nie zwiększa dokładności wyników. Co ciekawe, dla problemu o rozmiarze 323 zwiększenie rozmiaru populacji powyżej 250 pogorszyło jakość rozwiązań.

Prawdo-podobieństwo krzyżowania	0.6		0.7		0.8		0.85		0.9		0.95	
Rozmiar	Wynik	Błąd	Wynik	Błąd	Rezultat	Błąd	Wynik	Błąd	Wynik	Błąd	Rozmiar	Błąd
48	17458.9	0.211	15140.7	0.05	16125.7	0.118	15834	0.098	15961.1	0.107	17813.2	0.235
53	9677.8	0.402	7779.2	0.127	8343.6	0.208	8559.5	0.24	8432.3	0.221	11187.6	0.62
70	47470.6	0.227	40428.8	0.045	42478.9	0.098	42517.4	0.099	43454.8	0.124	49703.6	0.285
100	75926.4	1.096	43930.3	0.213	48496.9	0.339	48431.4	0.337	53491.7	0.476	87848.3	1.425
323	3366.6	1.539	2324.8	0.753	2427.6	0.831	2705.8	1.041	3126.6	1.358	3758.1	1.834
403	4610.3	0.87	3805.1	0.544	3938	0.598	4138.8	0.679	4435.4	0.799	5068.9	1.056
443	5043	0.854	4253.9	0.564	4400.3	0.618	4635.8	0.704	4918.7	0.808	5580.7	1.052

Table 3: Wyniki dla asymetrycznych danych o zadanej wielkości populacji dla prawdopodobieństwa mutacji 3% i prawdopodobieństwa krzyżowania 80%, maksymalnej ilości generacji 2500. Przedstawiony błąd to błąd względny.

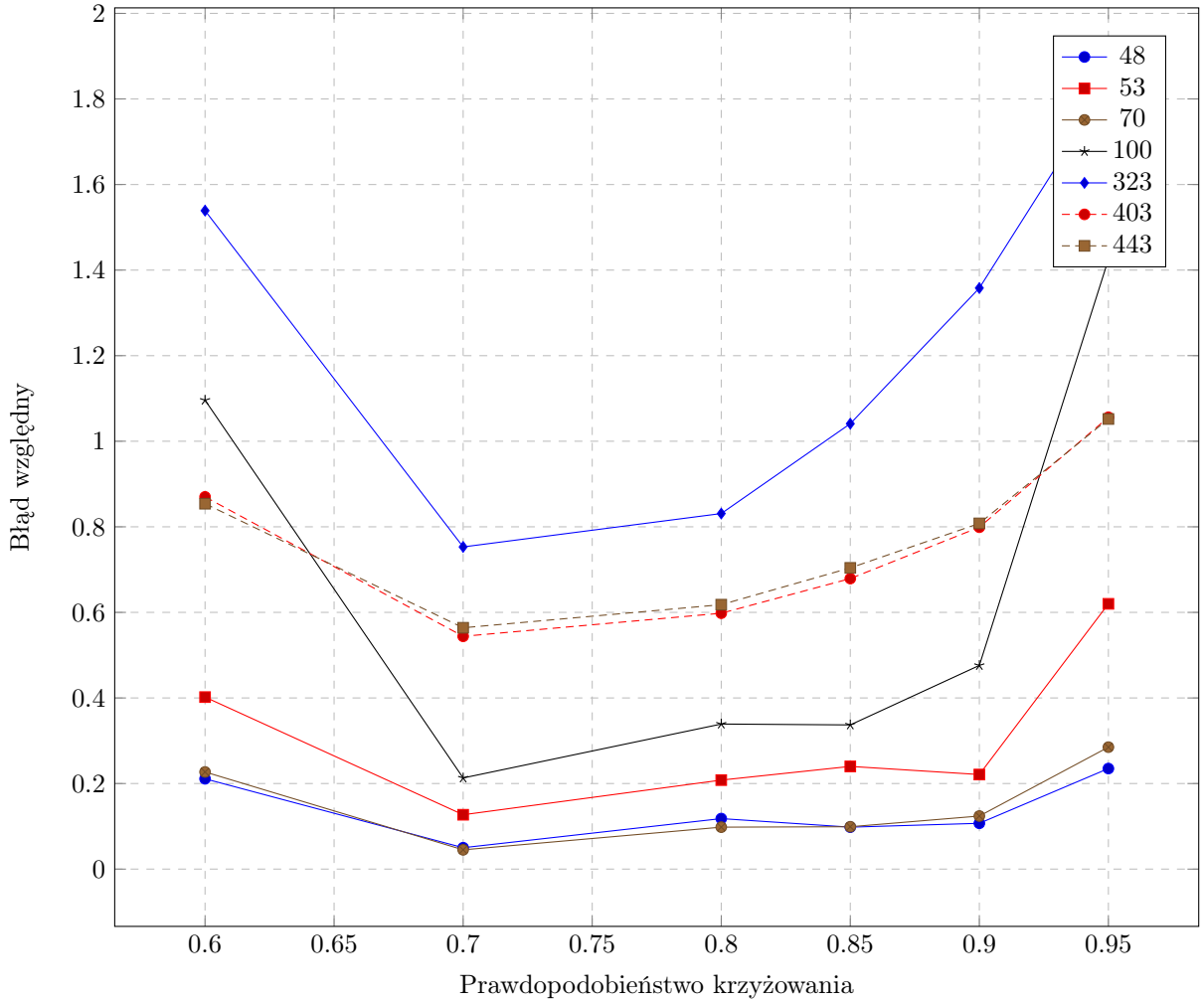

Figure 3: Błąd względny w zależności od prawdopodobieństwa krzyżowania dla prawdopodobieństwa mutacji 3%, rozmiaru populacji 250, maksymalnej ilości generacji 2500. Przedstawiony błąd to błąd względny.

Z wykresu można wywnioskować, że najlepszą wielkością parametru prawdopodobieństwa krzyżowania znajduje się gdzieś między wartościami 70% a 80%. Poniżej i powyżej tych wartości wartość błędu względnego gwałtownie rośnie. Prawdopodobieństwo krzyżowania nie może być zbyt wysokie, ponieważ tworzone jest zbyt wiele nowych osobników, z byt wiele 'silnych' rodziców zostaje straconych podczas generowania kolejnej populacji.

Prawdopodobieństwo mutacji	0.009		0.03		0.1		0.15		0.2		0.3	
Rozmiar	Wynik	Błąd	Wynik	Błąd	Rezultat	Błąd	Wynik	Błąd	Wynik	Błąd	Rozmiar	Błąd
48	15782	0.094	15669.5	0.086	16011.1	0.11	16832.4	0.167	29440.5	1.041	33495.4	1.323
53	8396	0.216	8351.5	0.209	8509	0.232	12551	0.818	15580.8	1.256	17395.7	1.519
70	42497.2	0.099	42450.6	0.098	42975.8	0.111	53415.7	0.381	55473.8	0.434	57850.4	0.496
100	50171.6	0.385	47338.5	0.307	50108.5	0.383	106063.5	1.928	121408.6	2.351	131980.2	2.643
323	2323.5	0.752	2468.3	0.861	3439.2	1.594	3989.4	2.009	4314.9	2.254	4792.4	2.614
403	3809.1	0.545	3941.4	0.599	4589.1	0.862	5084.1	1.063	5487.4	1.226	5904.4	1.395
443	4249	0.562	4412.5	0.622	5010.6	0.842	5549.2	1.04	6009.9	1.21	6466.8	1.378

Table 4: Wyniki dla asymetrycznych danych o zadanej wielkości populacji dla prawdopodobieństwa mutacji 3% i prawdopodobieństwa krzyżowania 80%, maksymalnej ilości generacji 2500. Przedstawiony błąd to błąd względny.

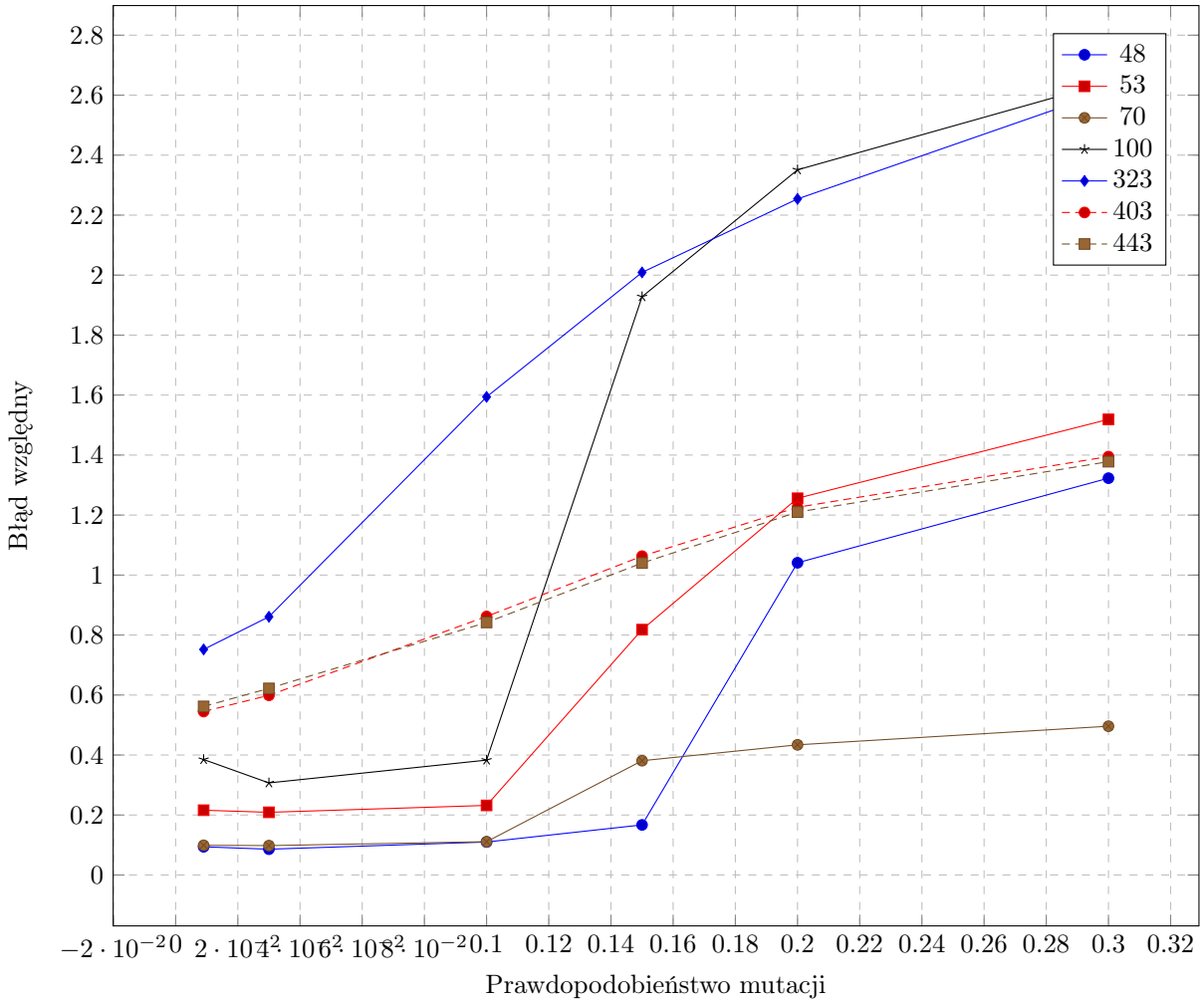

Figure 4: Błąd względny w zależności od prawdopodobieństwa mutacji dla prawdopodobieństwa krzyżowania 80%, rozmiaru populacji 250, maksymalnej ilości generacji 2500. Przedstawiony błąd to błąd względny.

Na powyższym wykresie widać, że dla dość niskich wartości prawdopodobieństwa mutacji wartość błędu względnego utrzymuje się niemal na stałym poziomie, chociaż zależność ta zależy od wybranej instancji problemu (dla instancji o rozmiarze 323 błąd względny rośnie już wraz ze wzrostem nawet małych wartości prawdopodobieństwa mutacji). Zadaniem mutacji jest wprowadzanie różnorodności w populacji i szersze przeszukiwanie przestrzeni rozwiązań. Dla zbyt wysokiego prawdopodobieństwa mutacji nie mamy szans odpowiednio dokładnej eksploatacji przestrzeni rozwiązań, niszcząc otrzymane już dobre rozwiązania.

5 Bibliografia

- `www.hindawi.com/journals/cin/2017/7430125/`
- `www.obitko.com/tutorials/genetic-algorithms/tsp-example.php`
- towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35
- `pl.wikipedia.org/wiki/Algorytm_genetyczny`
- `sound.eti.pg.gda.pl/student/isd/isd03-algorytmy_genetyczne.pdf`

Spis treści