Projektowanie Efektywnych Algorytmów

Kierunek *Informatyka*	**Termin** *Poniedziałek 15:15*
Temat *Algorytmy lokalnego przeszukiwania*	**Problem** *TSP*
Skład grupy *Kinga Marek 235 280*	**Nr grupy** -
Prowadzący *Mgr inż. Radosław Idzikowski*	**data** *December 24, 2019*

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 Symulowanie wyżarzanie

Symulowanie wyżarzanie to heurystyczny algorytm probabilistyczny, który jest wykorzystywany do przybliżania rozwiązania optymalnego w dużej przestrzeni rozwiązań. Wzorowany jest na zjawisku wyżarzania wykorzystywanego w metalurgii. Technika ta polega na podgrzewaniu materiału do wysokiej temperatury, a następnie na kontrolowanym i powolnym chłodzeniu. Podczas procesu chłodzenia formowana jest stała, krystaliczna struktura. Pojęcie powolnego chłodzenia w algorytmie wyżarzania odpowiada powolnemu spadkowi prawdopodobieństwa z jakim akceptowane są rozwiązania gorsze od obecnie znalezionego. Akceptowanie gorszych rozwiązań pozwala na szersze przeszukanie przestrzeni stanów i umożliwia znalezienie rozwiązania poza pierwszym znalezionym lokalnym minimum.

Dobór parametrów

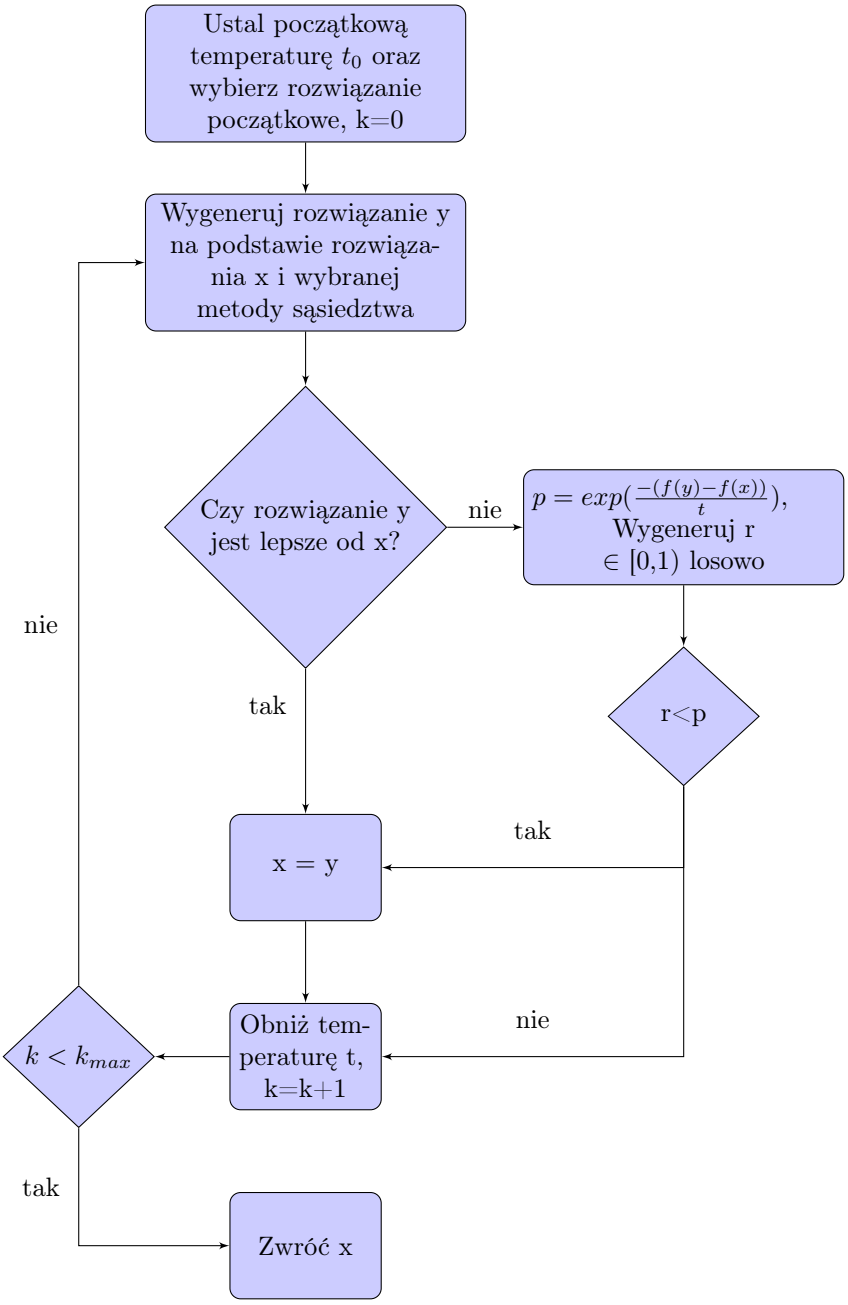
Potrzeba długiego i powolnego obniżania temperatury może nasunąć chęć ustalenia niskiej temperatury początkowej. Zbyt niska temperatura początkowa może jednak nie pozwolić atomom na odpowiednie uporządkowanie struktury. Poglądowe zależności przedstawiono poniżej[1].

$$\begin{aligned}
\uparrow \text{temperatura początkowa} &\rightarrow \text{czas optymalizacji} \uparrow \\
\downarrow \text{temperatura początkowa} &\rightarrow \text{energia końcowa} \uparrow \\
\uparrow \text{szybkość chłodzenia} &\rightarrow \text{energia końcowa} \uparrow \\
\downarrow \text{szybkość chłodzenia} &\rightarrow \text{czas optymalizacji} \uparrow
\end{aligned}$$

Chłodzenie powinno być na tyle wolne, by atomy materiału miały możliwość uformowania sieci krystalicznej z minimalną energią wewnętrzną. Konieczne jest więc znalezienie równowagi pomiędzy prędkością chłodzenia, a wysokością temperatury początkowej. Podczas implementacji wyżarzania dla wybranego problemu znaczącą rolę odgrywa odpowiedni dobór parametrów początkowych oraz algorytmów używanych do wyżarzania i generowania nowych rozwiązań.

[1] 'Investigation on the choice of the initial temperature in the Simulated Annealing: A Mushy State SA for TSP' H. Shakouri G, Kambiz Shojaee, M. Behnam T

Poszczególne kroki algorytmu



W pierwszym kroku algorytmu wybierana jest temperatura początkowa, początkowe rozwiązanie i licznik kroków ustalany jest na 0. Warunkiem stopu algorytmu jest wykonanie określonej ilości kroków k_{max}. W różnych implementacjach można się jednak spotkać z różnymi podejściami - jako warunkiem stopu może być tak także osiągniecie odpowiednio niskiej temperatury t lub upłyniecie określonej ilości czasu od wywołania początkowego algorytmu.
Rozwiązanie y generowane na podstawie dotychczasowego rozwiązania x także może być tworzone na wiele sposobów. Trzema najbardziej popularnymi i prostymi metodami są ruchy typu *swap*, *insert*, *reverse*. W każdej z metod losowane są dwa indeksy *index1* oraz *index2* wykorzystywane do modyfikacji obecnego rozwiązania. Dla metody *swap* pozycje na tych indeksach są zamieniane, a dla metody *reverse* kolejność pomiędzy *index1*, a *index2* jest odwracana. W metodzie *insert* elemnt na pozycji *index2* wpisywany jest na pozycję *index1*, a elementy za *index1* przesuwane są kolejno w prawo na pozycję n+1 aż do pozycji *index2* włącznie (przyjmujemy że *index1* $<$ *index2*).
Każde nowo znalezione rozwiązanie podlega ocenie i jeśli jest lepsze od obecnego, to zapisywane jest jako obecne rozwiązanie. Dla problemu komiwojażera porównywane są wartości funkcji kosztu (koszt przejścia

całej ścieżki komiwojażera). Jeśli nowe rozwiązanie jest gorsze od dotychczasowego to na podstawie funkcji prawdopodobieństwa akceptacji rozwiązania podejmowana jest decyzja, czy nowe, 'gorsze' rozwiązanie zostanie zapisane jako nowe, obecnie najlepsze rozwiązanie. Funkcja prawdopodobieństwa akceptacji nowo znalezionego rozwiązania:

$$p = \begin{cases} 1 & \text{dla: } f(y) \leq f(x) \\ e^{-\frac{(f(y)-f(x))}{t}} & \text{dla: } f(y) > f(x) \end{cases}$$

Jeszcze jednym podejściem charakterystycznym dla wybranej implementacji symulowanego wyżarzania jest dobór metody chłodzenia, czyli obniżania temperatury t. Temperatura może być w obniżana w sposób liniowy, wykładniczy lub logarytmiczny.

3 Implementacja

W drugim etapie projektu zostały zaimplementowane dwie wersje algorytmu wyżarzania. Pierwsza wykorzystuje podstawowe mechanizmy chłodzenia i wymaga ręcznego dostosowywania parametrów algorytmu. Druga została oparta na publikacji naukowej zatytułowanej "List-Based Simulated Annealing Algorithm for Traveling Salesman Problem". Implementację tą wyróżnia brak konieczności wybierania temperatury początkowej - przed rozpoczęciem wykonania algorytmu należy dobrać tylko ilość kroków jaką chcemy wykonać.

3.1 Podstawowa wersja algorytmu

W podstawowej wersji algorytmu użyto wykładniczego sposobu chłodzenia. Temperatura w każdej iteracji algorytmu była mnożona przez stałą $\alpha \in (0, 1)$ (zwykle α było bliskie 1). Temperatura w iteracji n można było obliczyć ze wzoru:

$$t_{n+1} = t_0 \cdot \alpha^n$$

Parametr *alpha*, początkowa temperatura oraz liczba kroków k_{max} podawane są przy początkowym wywołaniu algorytmu. Na listingu na kolejnej stronie można zobaczyć główną metodę wykonującą podstawy algorytm symulowanego wyżarzania napisaną w języku Go.

```
func (l *SimulatedAnnealing)
Resolve(steps int, temperatureStep float64, initialTemperature float64) ([]int, error){
        if l.AdjacencyMatrix == nil {
                return []int{},
                    errors.New("Adjacency Matrix not found! Initialize struct first!")
        }

        l.size = len(l.AdjacencyMatrix[0])
        solution := l.createInitialSolution()
        currentBestSolution := solution
        currentBestCost := CalculateCost(solution, l.AdjacencyMatrix)
        currentTemperature := initialTemperature

        var random float64
        var probability float64
        var newSolution []int
        var newCost int

        for k := 0; k<steps && currentTemperature > 0; k++ {

                index1 := rand.Intn(l.size)
                index2 := rand.Intn(l.size)
                newSolution = l.NeighboursGenerator
                    .GetSolutionFromNeighbourhood
                        (currentBestSolution, index1, index2)
                newCost = CalculateCost(newSolution, l.AdjacencyMatrix)

                        if newCost < currentBestCost {
                                currentBestCost = newCost
                                currentBestSolution = newSolution
                        } else {
                                random = rand.Float64()
                                probability = math
                                    .Exp(-float64(newCost-currentBestCost)
                                        /currentTemperature)
                                if random < probability {
                                        currentBestCost = newCost
                                        currentBestSolution = newSolution
                                }
                        }

                currentTemperature = temperatureStep * currentTemperature
        }

        return currentBestSolution, nil
}
```

3.2 Generowanie nowych rozwiązań

W celu poprawienia jakości algorytmu do generowania nowych rozwiązań w obu implementacjach algorytmu użyto metody składającej się z wielokrotnych ruchów. Dla raz wylosowanych indeksów *index1* i *index2* generowano trzy nowe rozwiązania wykorzystując operatory *swap*, *insert*, *reverse*. Następnie każde z trzech nowych rozwiązań poddawano ocenie i wybierano najlepsze z nich. Wykorzystany operator generowania nowych rozwiązań można opisać następująco:

$$\pi' = min(\text{inverse}(\pi, index1, index2), \text{swap}(\pi, index1, index2), \text{reverse}(\pi, index1, index2))$$

gdzie π' - nowe rozwiązanie, π - obecne rozwiązanie. Implementacja generatora załączona została na listingu poniżej.

```
type NeighboursGenerator interface {
        GetSolutionFromNeighbourhood(solution []int, index1 int, index2 int) []int
}

type MultipleMove struct{
        AdjacencyMatrix [][]int
}

func (m MultipleMove)
    GetSolutionFromNeighbourhood(solution []int, index1 int, index2 int) []int{
        insertSolution := Insert{}
            .GetSolutionFromNeighbourhood
                (append([]int(nil),solution...), index1, index2)
        reverseSolution := Reverse{}
            .GetSolutionFromNeighbourhood
                (append([]int(nil),solution...), index1, index2)
        swapSolution := Swap{}
            .GetSolutionFromNeighbourhood
                (append([]int(nil),solution...), index1, index2)

        bestSolution := insertSolution

        if CalculateCost(insertSolution, m.AdjacencyMatrix)
            > CalculateCost(reverseSolution, m.AdjacencyMatrix) {
                bestSolution = reverseSolution
        }
        if CalculateCost(bestSolution, m.AdjacencyMatrix)
            > CalculateCost(swapSolution, m.AdjacencyMatrix) {
                bestSolution = swapSolution
        }

        return bestSolution
}
```

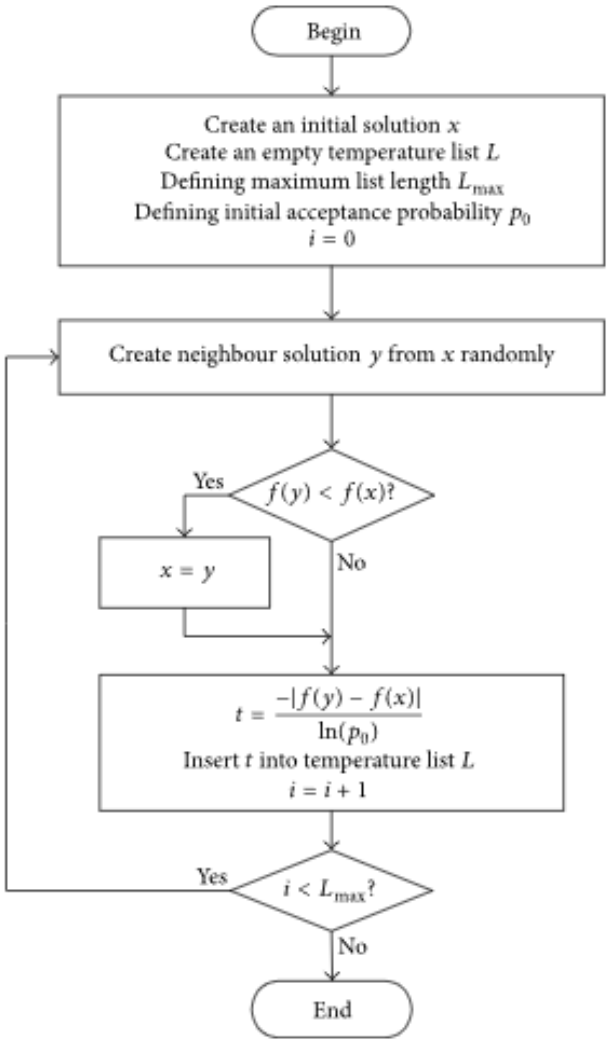
W języku Go struktury *slice* w których przechowywane jest aktualne rozwiązanie *solution* przekazywane są przez referencje przez co w każdym wywołaniu operatorów koniecznie było wywołanie funkcji *append*, która kopiuje elementy ze z jednego *slice* do drugiego i zwraca nowy (w miejscu wywołania funkcji tworzymy w kopię *solution*).
Dla uniezależnienia się od określonej metody generowania rozwiązania w sąsiedztwie obecnego stworzono oddzielny interfejs *NeighboursGenerator*, który zawierała każda ze struktur implementująca wyżarzanie (zarówno podstawowa wersja i i list-based omówiona w kolejnym punkcie). Dla uproszczenia na listingu pokazano tylko implementację operatora bazującego na pozostałych trzech podstawowych. Każdy z czterech operatorów generowania nowych rozwiązań implementuje interfejs *NeighboursGenerator* przez co może być w łatwy sposób używany w strukturach *SimulatedAnnealing* oraz *ListBasedSimulatedAnnealing* (wersja podstawowa i list-based algorytmu wyżarzania).

3.3 List-based simulated annealing

Podczas przeprowadzania eksperymentów na podstawowej wersji algorytmu zauważono wrażliwość algorytmu na dobór parametrów takich jak temperatura początkowa lub sposób chłodzenia. W celu uniknięcia problemów związanych z doborem tych parametrów zaimplementowano wyżarzanie bazujące na liście. Implementacja opiera się na publikacji Shi-hua Zhan, Juan Lin, Ze-jun Zhang oraz Yi-wen Zhong zatytułowanej "List-Based Simulated Annealing Algorithm for Traveling Salesman Problem". W ich publikacji opisano także operator generowania nowych rozwiązań bazujący na trzech pozostałych opeartorach *swap*, *insert*, *reverse*, który został wykorzystany też w algorytmie podstawowym opisanym w punkcie 3.1.

Algorytm wyżarzania bazujący na liście w pierwszym kroku generuje listę temperatur, których wartości uzależnione są od przestrzeni stanów wybranego problemu. W pierwszym etapie algorytmu

generowana jest lista temperatur o rozmiarze określonym rozmiarze, która w kolejnych krokach będzie aktualizowana na podstawie różnic w energiach pomiędzy nowymi i obecnymi rozwiązaniami.



W kolejnych krokach algorytmu z listy zdejmowana jest maksymalna temperatura i jest używana do ewaluacji funkcji prawdopodobieństwa akceptacji nowego rozwiązania. Podejście list-based wyróżnia określona ilość powtórzeń generowania nowych rozwiązań dla danej temperatury. Temperatura nie zmienia się co iterację, lecz co wybraną ilość kroków. Za każdym razem, gdy dla danej temperatury akceptowane jest rozwiązanie gorsze od obecnego obliczana jest nowa temperatura, których suma przechowywana w zmiennej lokalnej pętli kroków dla danej temperatury. Po wykonaniu określonej liczby kroków dla danej temperatury liczona jest średnia ze zsumowanych temperatur i lista temperatur jest aktualizowana. W celach optymalizacyjnych jako kontenera na temperatury użyto kolejki priorytetowej, a nie listy co może sugerować nazwa algorytmu. Częste znajdowanie temperatury o największej wartości byłoby bardziej kosztowne dla listy niż dla kolejki priorytetowej. Pełny przebieg algorytmu przedstawiono na diagramie poniżej.

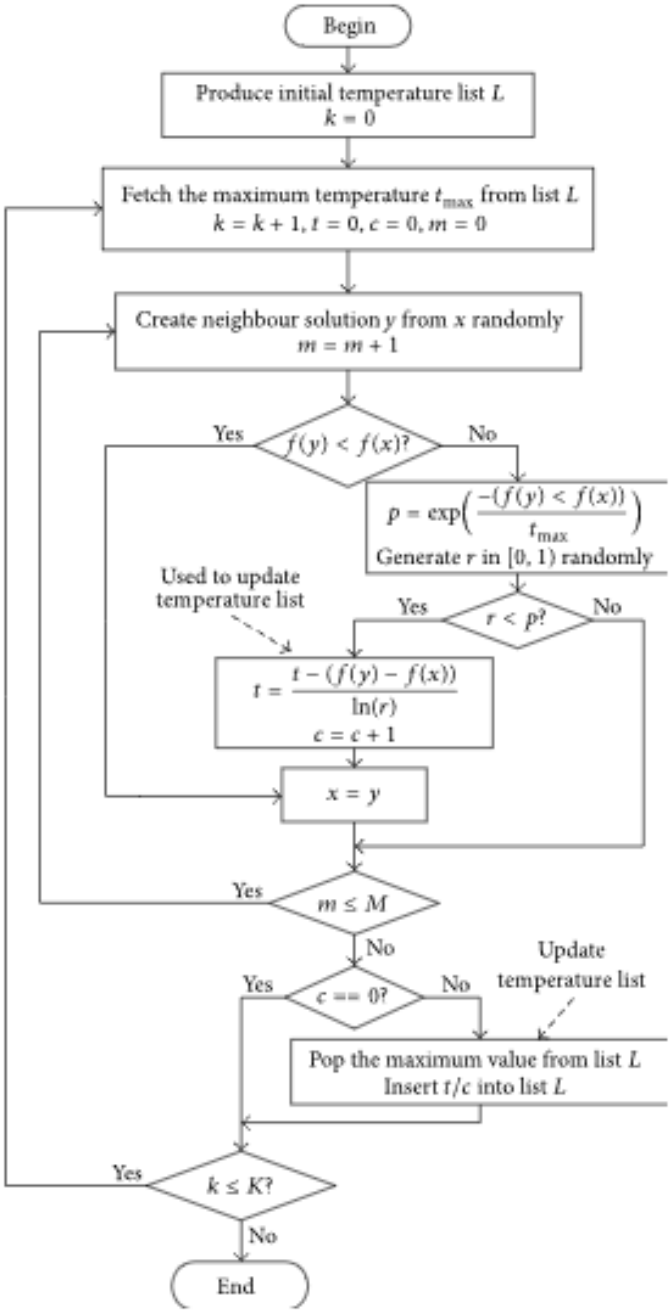


Diagram ten pochodzi bezpośrednio z publikacji "List-Based Simulated Annealing Algorithm for Traveling Salesman Problem". Pojawił się w nim jednak jeden błąd - wewnątrz bloku w 'Used to update temperature list' zmienna t przechowuje sumę wszystkich temperatur wygenerowanych podczas każdej akceptacji gorszego rozwiązania tak więc $t = t + \frac{-|f(y)-f(x)|}{ln(r)}$. Implentacja algorytmu list-based w języku Go została przedstawiona poniżej.

```
func (l *ListBasedSimulatedAnnealing) Resolve(steps int) ([]int, error){
        repeatTemperature := 200

        if l.AdjacencyMatrix == nil {
                return []int{}, errors.New
                  ("Adjacency Matrix not found! Initialize struct first!")
        }

        l.size = len(l.AdjacencyMatrix[0])
        solution := l.createInitialSolution()
        currentBestSolution := solution
        currentBestCost := CalculateCost(solution, l.AdjacencyMatrix)
        temperatures := l.setInitialTemperatureList(solution)

        var currentTemperature float64
        var random float64
        var probability float64
        var temperaturesSum float64
        var acceptedWorseSolutionCount int
        var newSolution []int
        var newCost int

        for k := 0; k<steps && temperatures.Len() > 0; k++ {
                temperaturesSum = 0.0
                acceptedWorseSolutionCount = 0
                currentTemperature,_ = l.popMaxTemperature(temperatures)

                for m := 0; m < repeatTemperature; m++ {
                        newSolution, newCost = l.generateNewSolutionAndCost
                            (currentBestSolution)

                        if newCost < currentBestCost {
                                currentBestCost = newCost
                                currentBestSolution = newSolution
                        } else {
                                random = rand.Float64()
                                probability = math.Exp(-float64(newCost-currentBestCost)
                                    / currentTemperature)
                                if random < probability {
                                        acceptedWorseSolutionCount =
                                            acceptedWorseSolutionCount + 1
                                        temperaturesSum = temperaturesSum +
                                            (-float64(newCost-currentBestCost)
                                            / math.Log(random))
                                        currentBestCost = newCost
                                        currentBestSolution = newSolution
                                }
                        }
                }

                if acceptedWorseSolutionCount > 0 {
                        temperatures.Pop()
                        newTemperature := temperaturesSum /
                            float64(acceptedWorseSolutionCount)
                        temperatures.Insert(newTemperature, -newTemperature)
                }
        }
        return currentBestSolution, nil
```

```
}
```

Metodę starano się podzielić na mniejsze części takie jak *setInitialTemperatureList*, *popMaxTemperature*, czy *generateNewSolutionAndCost*. Do przechowywania temperatur użyto kolejki priorytetowej [github.com/jupp0r/go-priority-queue], która dla zapewnienia generyczności używała interfejsów (język Go nie wspiera bezpośrednio programowania generycznego, jest to możliwe tylko pośrednio przez użycie interfejsów - każda metoda i typ implementują pusty interfejs). Jednak odczyt danych po zapisie ich do struktury takie przechowującej typ *interface{}* okazało się dość trudne i konieczne było użycie mechanizmu refleksji. Przykład użycia pokazano poniżej w metodzie *popMaxTemperature*.

```
func (l *ListBasedSimulatedAnnealing)
    popMaxTemperature(temperatures queue.PriorityQueue) (float64, error) {
        var floatInterface interface{}
        floatInterface, _ = temperatures.Pop()
        reflected := reflect.ValueOf(floatInterface)
        if reflected.IsValid() {
                currentTemperature := float64(reflected.Float())
                return currentTemperature, nil
        } else {
                return 0.0, errors.New("Could not read from queue!")
        }
}
```

Ostatnią wartą uwagi metodą może być generowanie początkowej listy temperatur oparte na pierwszym diagramie przedstawionym w punkcie 3.3.

```
func (l ListBasedSimulatedAnnealing)
    setInitialTemperatureList(initialSolution []int) queue.PriorityQueue {
    var newSolution []int
        var newCost int
    var newTemperature float64
    temperaturesQueue := queue.New()
    currentBestSolution := initialSolution
    currentBestCost := CalculateCost(currentBestSolution, l.AdjacencyMatrix)
        listLength := 250
        initialProbability := 0.9

        for i:=0; i<listLength; i++ {
                index1 := rand.Intn(l.size)
                index2 := rand.Intn(l.size)
                newSolution = l.NeighboursGenerator.
                    GetSolutionFromNeighbourhood
                        (currentBestSolution, index1, index2)
                newCost = CalculateCost(newSolution, l.AdjacencyMatrix)

                if newCost < currentBestCost {
                        currentBestCost = newCost
                        currentBestSolution = newSolution
                } else {
                        newTemperature = -(float64(newCost-currentBestCost))
                            /math.Log(initialProbability)
                        temperaturesQueue.Insert(newTemperature, -newTemperature)
                }
        }

        return temperaturesQueue
}
```

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).

Rozmiar	Koszt optymalny	100 powtórzeń		150 powtórzeń		200 powtórzeń		300 powtórzeń		400 powtórzeń	
		Wynik	Błąd	Wynik	Błąd	Wynik	Błąd	Rezultat	Błąd	Wynik	Błąd
34	1286	1619	0.259	1585	0.233	1564	0.216	1536	0.194	1529	0.189
36	1473	1846	0.253	1792	0.217	1760	0.195	1728	0.173	1721	0.168
39	1530	1927	0.259	1885	0.232	1843	0.205	1821	0.190	1820	0.190
43	5620	5649	0.005	5643	0.004	5639	0.003	5637	0.003	5634	0.002
45	1613	2251	0.396	2160	0.339	2108	0.307	2081	0.290	2079	0.289
48	14422	17013	0.180	16597	0.151	16307	0.131	15974	0.108	15795	0.095
53	6905	9805	0.420	9282	0.344	9138	0.323	8836	0.280	8658	0.254
56	1608	2532	0.575	2414	0.501	2341	0.456	2263	0.407	2215	0.377
65	1839	3144	0.710	3026	0.645	2923	0.589	2816	0.531	2743	0.492
70	38673	46200	0.195	45435	0.175	44862	0.160	44174	0.142	43736	0.131
71	1950	3565	0.828	3373	0.730	3282	0.683	3141	0.611	3063	0.571
100	36230	64154	0.771	60419	0.668	58042	0.602	55049	0.519	53376	0.473
171	2755	10115	2.672	9500	2.448	9059	2.288	8518	2.092	8229	1.987
323	1326	3323	1.506	3270	1.466	3239	1.443	3181	1.399	3165	1.387
358	1163	3667	2.153	3577	2.076	3503	2.012	3467	1.981	3427	1.947
403	2465	4628	0.877	4535	0.840	4505	0.828	4431	0.798	4402	0.786
443	2720	5144	0.891	5055	0.858	5014	0.843	4946	0.818	4938	0.815

Table 1: Wyniki dla asymetrycznych danych o zadanym rozmiarze dla różnej ilości powtórzeń dla tej samej temperatury dla algorytmu list-based oraz błąd względny dla otrzymanych wyników

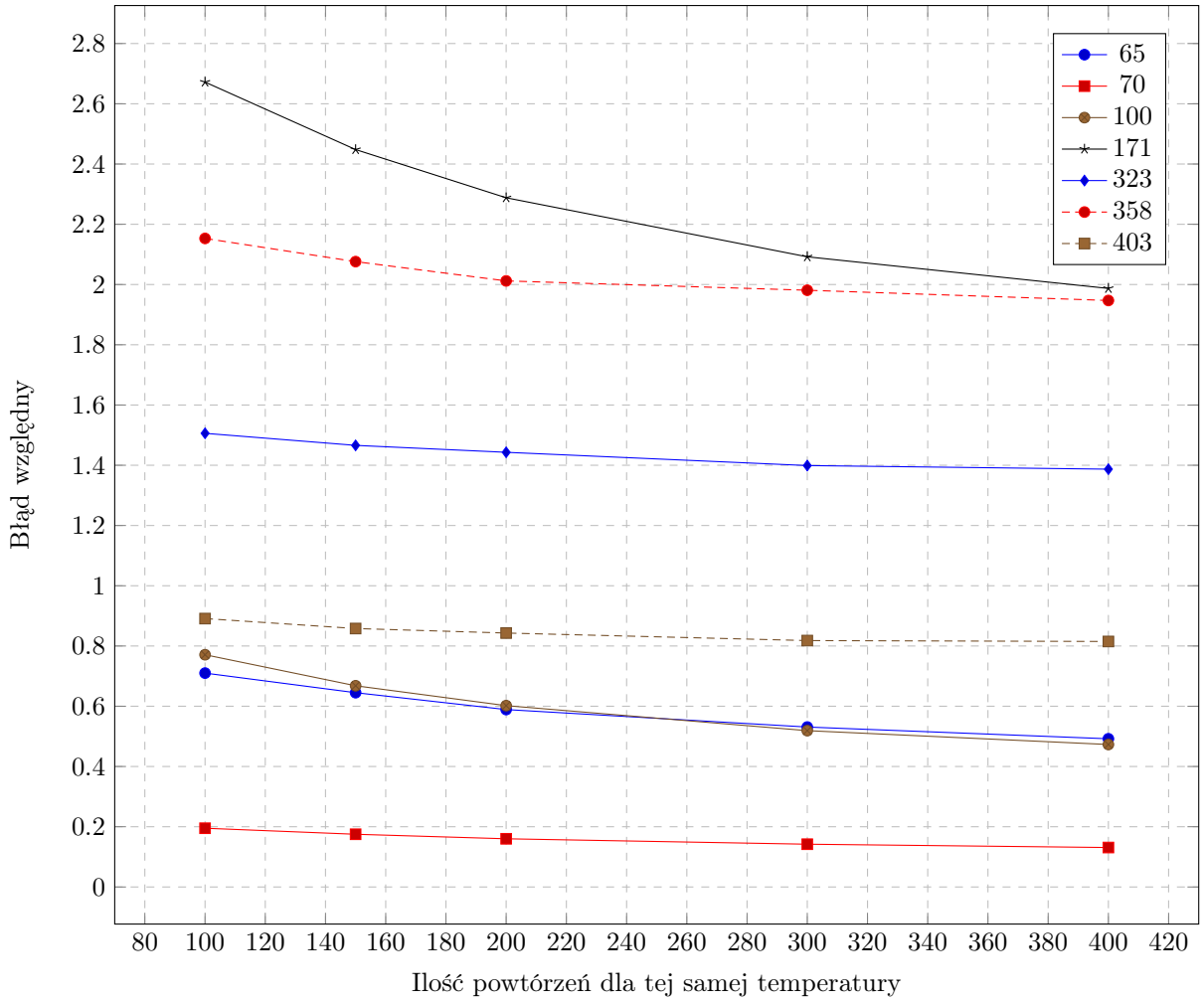


Figure 1: Błąd względny w zależności od liczby powtórzeń iteracji dla tej samej temperatury w algorytmie list-based

Na wykresie można zobaczyć, że liczba powtórzeń dla danej temperatury ma niewielki wpływ na działanie algorytmu. Można też zauważyć, że błąd względny jest niezależny od rozmiaru problemu, lecz zależy bardziej od struktury przestrzeni stanów wybranej instancji.

Rozmiar	Koszt optymalny	10 elementów		50 elementów		100 elementów		150 elementów		200 elementów	
		Wynik	Błąd	Wynik	Błąd	Wynik	Błąd	Rezultat	Błąd	Wynik	Błąd
34	1286	2247	0.747	1834	0.426	1679	0.306	1640	0.275	1586	0.233
36	1473	2512	0.705	2078	0.411	1928	0.309	1844	0.252	1785	0.212
39	1530	2711	0.772	2224	0.454	2046	0.337	1937	0.266	1886	0.233
43	5620	6460	0.149	5692	0.013	5661	0.007	5648	0.005	5640	0.004
45	1613	3230	1.002	2604	0.614	2369	0.469	2235	0.386	2171	0.346
48	14422	23570	0.634	18830	0.306	17539	0.216	16983	0.178	16573	0.149
53	6905	13859	1.007	11143	0.614	10224	0.481	9652	0.398	9288	0.345
56	1608	3955	1.460	3066	0.907	2714	0.688	2561	0.593	2440	0.517
65	1839	4820	1.621	3813	1.073	3395	0.846	3163	0.720	3024	0.644
70	38673	53952	0.395	48676	0.259	46988	0.215	45890	0.187	45361	0.173
71	1950	5408	1.773	4263	1.186	3794	0.946	3538	0.814	3377	0.732
100	36230	99121	1.736	75536	1.085	66985	0.849	62976	0.738	60406	0.667
171	2755	15975	4.799	12412	3.505	10976	2.984	10154	2.686	9542	2.464
323	1326	4880	2.680	4036	2.044	3637	1.743	3433	1.589	3279	1.473
358	1163	5471	3.704	4410	2.792	3984	2.426	3733	2.210	3585	2.083
403	2465	6273	1.545	5303	1.151	4907	0.991	4699	0.906	4551	0.846
443	2720	6950	1.555	5884	1.163	5462	1.008	5215	0.917	5081	0.868

Table 2: Wyniki dla asymetrycznych danych o zadanym rozmiarze dla różnej wielkości początkowej listy temperatur przy stałej ilośći powtórzeń dla danej temperatury (200) dla algorytmu list-based oraz błąd względny dla otrzymanych wyników

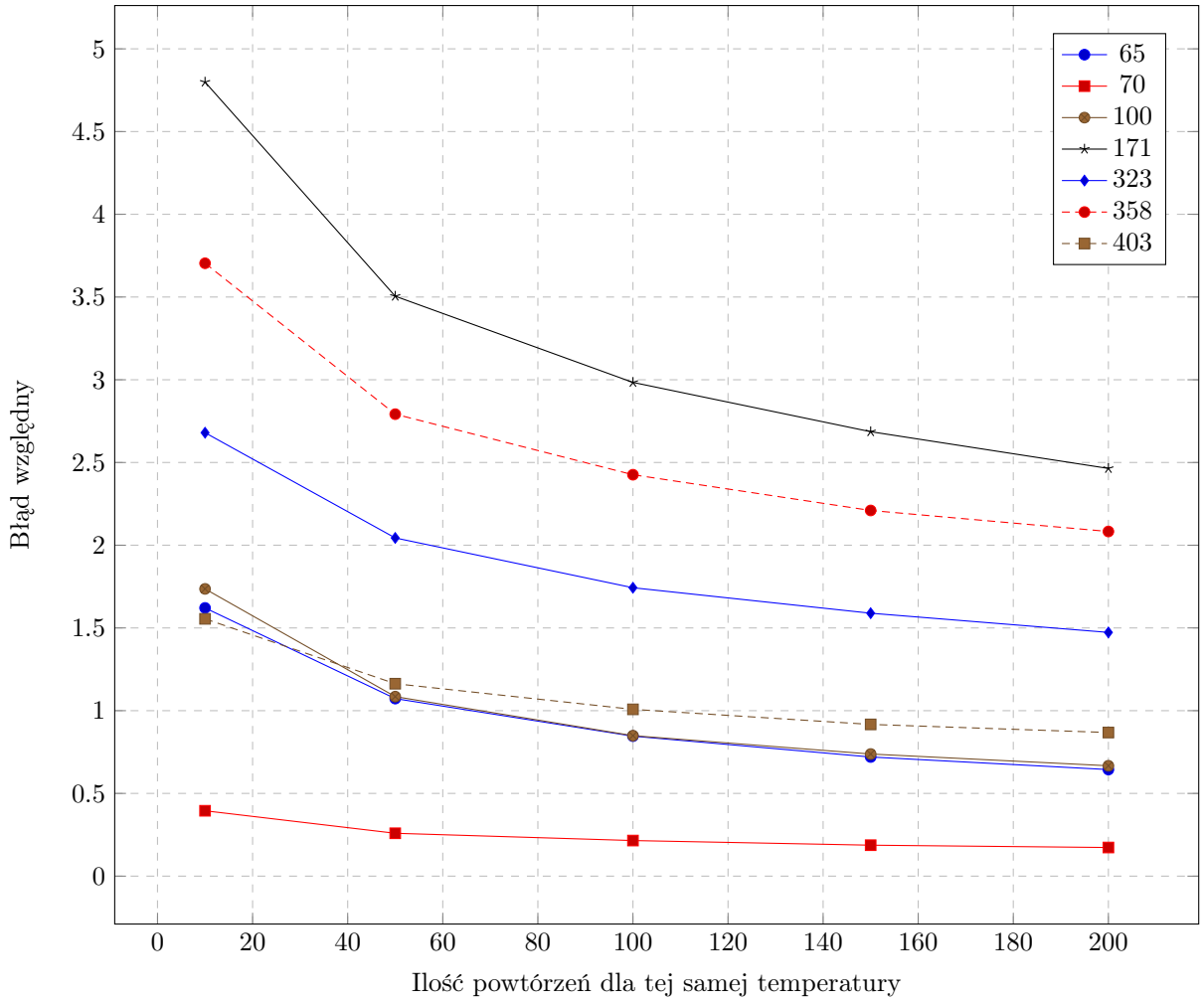


Figure 2: Błąd względny w zależności od wielkości początkowej listy temperatur i w algorytmie list-based dla różnych wielkości instancji

Porównując Figure 2 do Figure 1 można wywnioskować, że na jakość działania algorytmu list-based większe znaczenie ma dobranie dobrej wielkości początkowej liczby temperatur generowanej na początku algorytmu niż ilość powtórzeń dla danej temperatury. Wykres ten w mniejszym stopniu przbliżony jest do stałej niż na wykresie Figure 2.

Rozmiar	α =0.99		α =0.995		α =0.997		α =0.999		α =0.9994		α =0.9996	
	Wynik	Błąd	Wynik	Błąd	Wynik	Błąd	Rezultat	Błąd	Wynik	Błąd	Wynik	Błąd
34	1489	0.158	1502	0.168	1471	0.144	1480	0.151	1519	0.181	1818	0.414
36	1655	0.124	1665	0.130	1665	0.130	1663	0.129	1688	0.146	2021	0.372
39	1730	0.131	1726	0.128	1728	0.129	1748	0.142	1787	0.168	2185	0.428
43	5626	0.001	5625	0.001	5625	0.001	5627	0.001	5666	0.008	5900	0.050
45	1912	0.185	1938	0.201	1918	0.189	1921	0.191	1982	0.229	2482	0.539
48	15549	0.078	15593	0.081	15498	0.075	15476	0.073	15422	0.069	15559	0.079
53	8973	0.299	8866	0.284	8851	0.282	8566	0.241	8563	0.240	8869	0.284
56	2017	0.254	2007	0.248	2008	0.249	2030	0.262	2174	0.352	2872	0.786
65	2441	0.327	2451	0.333	2445	0.330	2507	0.363	2674	0.454	3505	0.906
70	43435	0.123	43582	0.127	43504	0.125	43340	0.121	43398	0.122	43832	0.133
71	2644	0.356	2641	0.354	2663	0.366	2727	0.398	2948	0.512	3870	0.985
100	50461	0.393	49699	0.372	49780	0.374	49233	0.359	48844	0.348	50251	0.387
171	6769	1.457	6786	1.463	6838	1.482	7256	1.634	8022	1.912	10325	2.748
323	2323	0.752	2349	0.771	2380	0.795	2616	0.973	3261	1.459	4813	2.630
358	2440	1.098	2470	1.124	2501	1.150	2783	1.393	3504	2.013	5245	3.510
403	3587	0.455	3613	0.466	3651	0.481	3912	0.587	4591	0.862	6057	1.457
443	4025	0.480	4057	0.492	4092	0.504	4366	0.605	5087	0.870	6621	1.434

Table 3: Wyniki dla asymetrycznych danych o podanej szybkości chłodzenia α dla algorytmu podstawowego i temperatury początkowej $t_0 = 10000$. Przedstawiony błąd to błąd względny.

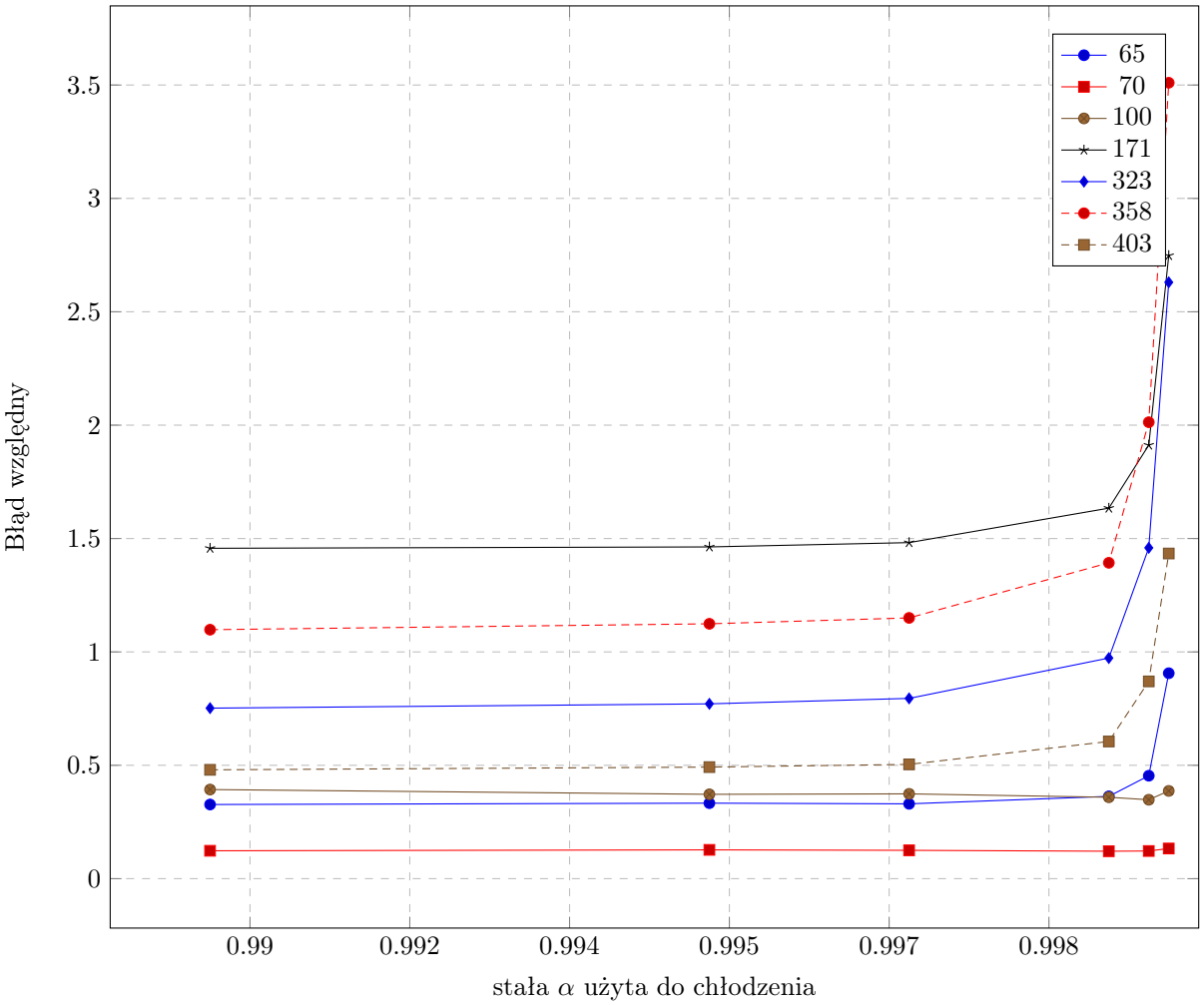


Figure 3: Błąd względny w zależności od szybkości chłodzenia α w algorytmie podstawowym dla różnych wielkości instancji i temperatury początkowej $t_0 = 10000$

Na wykresie tym można zobaczyć szybki spadek jakości algorytmu dla temperatury chłodzenia większej niż 0.998. Zbyt szybkie chłodzenie nie pozwala na odpowiednio głębokie przeszukiwanie przestrzeni stanów i algorytm nie jest w stanie wyjść poza znalezione lokalne minimum (funkcja akceptacji gorszego rozwiązania dla niskich temperatur akceptuje gorsze rozwiązania z bardzo niskim prawdopodobieństwem).

Rozmiar	t_0 =400		t_0 =700		t_0 =1000		t_0 =1300		t_0 =1600		t_0 =2000	
	Wynik	Błąd	Wynik	Błąd	Wynik	Błąd	Rezultat	Błąd	Wynik	Błąd	Wynik	Błąd
34	1472	0.145	1483	0.153	1471	0.144	1472	0.145	1484	0.154	1476	0.148
36	1658	0.126	1676	0.138	1663	0.129	1653	0.122	1662	0.128	1669	0.133
39	1714	0.120	1727	0.129	1763	0.152	1743	0.139	1752	0.145	1752	0.145
43	5625	0.001	5627	0.001	5626	0.001	5627	0.001	5627	0.001	5627	0.001
45	1907	0.182	1915	0.187	1936	0.200	1919	0.190	1931	0.197	1934	0.199
48	15431	0.070	15430	0.070	15506	0.075	15490	0.074	15571	0.080	15431	0.070
53	8528	0.235	8556	0.239	8615	0.248	8607	0.246	8568	0.241	8647	0.252
56	2027	0.261	2039	0.268	2071	0.288	2051	0.275	2067	0.285	2088	0.299
65	2470	0.343	2485	0.351	2508	0.364	2513	0.367	2512	0.366	2554	0.389
70	43105	0.115	43266	0.119	43408	0.122	43380	0.122	43304	0.120	43409	0.122
71	2694	0.382	2722	0.396	2777	0.424	2767	0.419	2770	0.421	2768	0.419
100	49217	0.358	49355	0.362	49007	0.353	49339	0.362	49417	0.364	49302	0.361
171	7033	1.553	7134	1.589	7239	1.628	7299	1.649	7388	1.682	7439	1.700
323	2537	0.913	2575	0.942	2607	0.966	2637	0.989	2663	1.008	2681	1.022
358	2685	1.309	2743	1.359	2774	1.385	2802	1.409	2840	1.442	2869	1.467
403	3824	0.551	3861	0.566	3907	0.585	3938	0.598	3957	0.605	3990	0.619
443	4265	0.568	4327	0.591	4357	0.602	4397	0.617	4419	0.625	4450	0.636

Table 4: Wyniki dla asymetrycznych danych o podanej temperaturze początkowej t_0 dla algorytmu podstawowego i prędkości chłodzenia $\alpha = 0.999$. Przedstawiony błąd to błąd względny.

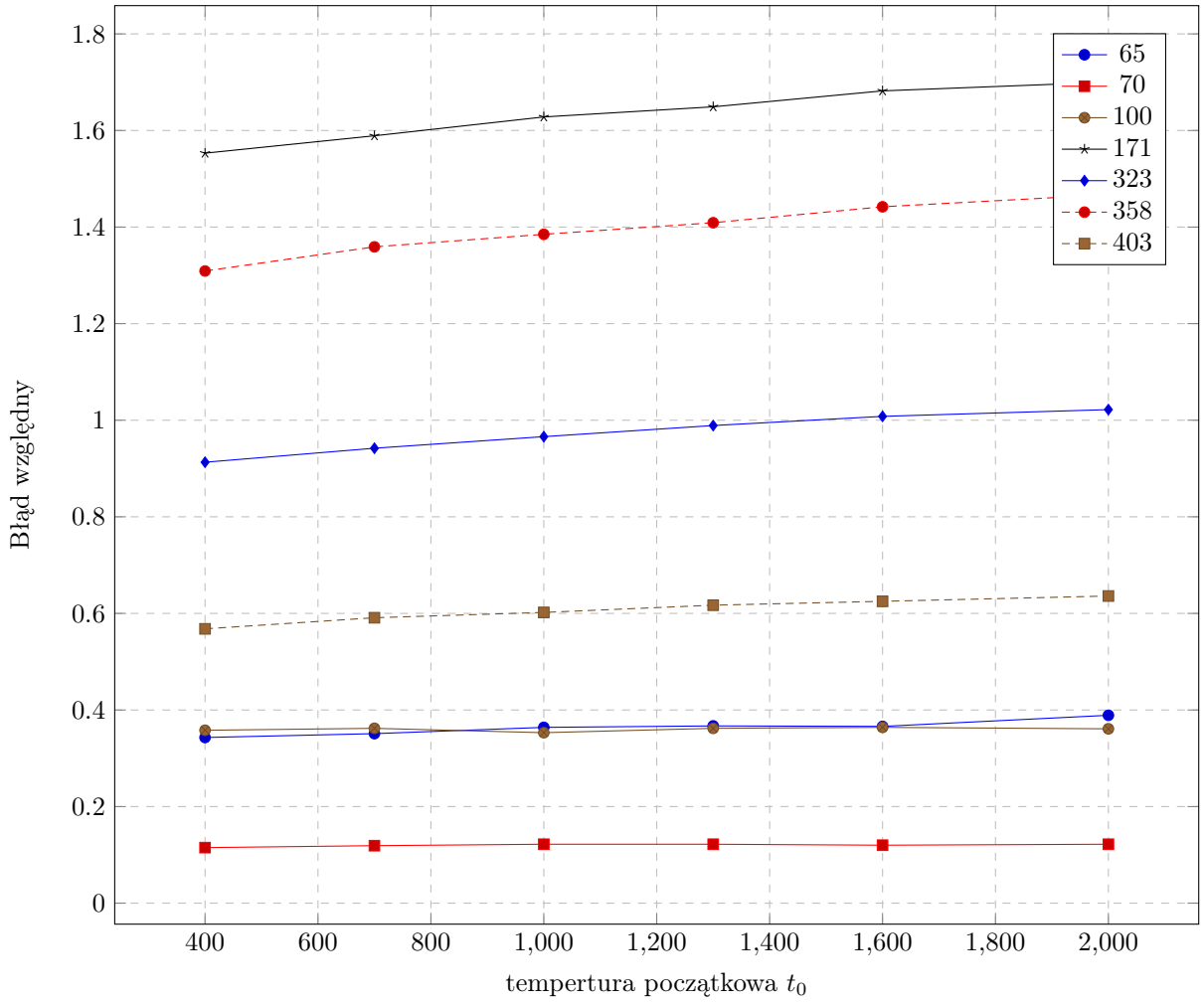


Figure 4: Błąd względny w zależności od temperatury początkowej w algorytmie podstawowej dla stałej prędkości chłodzenia $\alpha = 0.999$

Algorytm podstawowy jest zależny od wielkości początkowej temperatury jednak w bardziej zrównoważony sposób niż jak dla wielkości α sterującej szybkością chłodzenia. Wysoka temperatura początkowa może wymagać dłuższej ilości kroków, ponieważ przez dłuższy algorytm będąc w wysokiej temperaturze akceptuje gorsze rozwiązania, czyniąc swoje działanie bardziej losowym.

5 Wnioski

Wnioski do poszczególnych wykresów zostały zamieszczone bezpośrednio pod nimi.
W projekcie dużym rozczarowaniem okazała się implementacja algorytmu wyżarzania typu list-based według publikacji 'List-Based Simulated Annealing Algorithm for Traveling Salesman Problem'. Celem wprowadzenia początkowej listy temperatur miała być eliminacja konieczności dostosowywania parametrów takich jak temperatura początkowa i prędkość chłodzenia. Jakość rozwiązań jednak nadal zależała od doboru parametrów jakimi posługiwał się algorytm (ilość powtórzeń dla danej temperatury i długość początkowej listy temperatur), a sam algorytm dawał wyniki podobne lub nawet niewiele gorsze od tych otrzymanych w wersji podstawowej.

6 Bibliografia

- Publikacja 'List-Based Simulated Annealing Algorithm for Traveling Salesman Problem' `www.hindawi.com/journals/cin/2016/1712630/#B25`
- `https://hal.archives-ouvertes.fr/hal-01962049/document`
- `en.wikipedia.org`
- 'Investigation on the choice of the initial temperature in the Simulated Annealing: A Mushy State SA for TSP', H. Shakouri G, Kambiz Shojaee, M. Behnam T (publikacja pobrana za pomocą SciHuba)

Spis treści