Arkadiusz Kozak 226122 **21.12.2017**

Prowadzący: **dr Inż. Dariusz Banasiak**

Algorytm genetyczny dla problemu komiwojażera

1. Opis problemu:

Zadanie polega na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Danych mamy ***N*** miast które komiwojażer ma odwiedzić oraz odległości między nimi. Celem jest znalezienie najkrótszej ścieżki łączącej wszystkie miasta i kończącej się w określonym punkcie.

2. Opis algorytmu:

Algorytm genetyczny, zwany też ewolucyjnym, należy do grupy algorytmów metaheurystycznych i został opracowany przez Johna Hollanda. Zauważył on podobieństwo między procesem ewolucji a problemami optymalizacji. Ewolucja bowiem również jest procesem, w którym natura "poszukuje" rozwiązań najlepszych dla danych warunków. Można zatem posłużyć się pewną przenośnią w opisie algorytmu i wprowadzić takie pojęcia jak "krzyżowanie", "mutacja", "osobniki", "populacja" czy "pokolenie". Algorytm w każdej iteracji poszukuje osobników (permutacji) lepszych pod względem funkcji celu. Iteracje natomiast można nazwać "pokoleniem", które wydało na świat nowe osobniki i tylko te lepiej przystosowane (lepsze pod względem funkcji celu) zostaną przyjęte przez algorytm do nowej populacji.

Algorytm genetyczny jest, podobnie jak tabu search, a w przeciwieństwie do symulowanego wyżarzania, algorytmem deterministycznym, co oznacza, że czas jego działania jest z góry określony przez dokonującego testu. Stanowi to pewnego rodzaju zaletę, ponieważ w niektórych przypadkach bardziej niż jakość rozwiązania liczy się czas, który może być ściśle ograniczony.

2.1 Krzyżowanie:

Zaimplementowany program korzysta w krzyżowania typu OX. Jak pozostałe typy, bazuje ono na pobieraniu pewnej części permutacji z rodziców R1 oraz R2, a następnie przekazywaniu ich potomstwu. W przedstawionym projekcie z dwójki rodziców powstaje dwóch potomków. Algorytm krzyżowania OX działa w następujący sposób:

- Z rodzica R1 bierzemy losowy przedział z zakresu od 0 do długości rodzica R1 i wpisujemy go do potomka w te same komórki
- Następnie z rodzica R2, rozpoczynając od prawej krawędzi umieszczonego przedziału umieszczamy kolejno elementy, nie znajdujące się jeszcze w potomku. Po umieszczeniu elementu pod ostatnim indeksem permutacji po prostu wracamy na początek R2 i kontynuujemy.

Aby otrzymać dwóch potomków, zamieniamy rodziców R1 i R2 miejscami.

2.2 Mutacje:

W projekcie użyta została metoda mutacji poprzez inwersje. Działa to w taki sposób, że pobierając danego osobnika R, wybieramy w nim dwa losowe indeksy i zamieniamy je miejscami. Jako parametr mutacja przyjmuje prawdopodobieństwo zajścia mutacji oraz ilość mutacji które zachodzą.

3. Parametry algorytmu:

Parametrami istotnymi dla działania algorytmu są:

K – ilość iteracji (czyli wygenerowanych pokoleń)

L – rozmiar populacji

P – prawdopodobieństwo mutacji

M – ilość przeprowadzonych jednorazowo mutacji

Pw – populacja wejściowa

Pn – nowa populacja

Pk – populacja końcowa

Ilość iteracji ma bezpośredni wpływ na ewoluowanie osobników, jednak zbyt duże wartości nie przekładają się w znaczący sposób na znajdywanie lepszych permutacji. Przekładają się one jednak zauważalnie na czas pracy algorytmu. Dlatego też zamiast ustawiać zbyt dużej ilości iteracji, warto zwiększyć nieco prawdopodobieństwo mutacji osobników, ponieważ istnieje wtedy spora szansa na przejście do innej przestrzeni rozwiązań i uniknięcia problemów minimum lokalnego. Wielkość populacji również ma wpływ na wynik, jednak można łatwo zauważyć że po znalezieniu wielkości bliskiej optymalnej jej dalsze zwiększanie (nawet gwałtowne), nie przekłada się w żaden sposób na jakość rozwiązania. Dodana także została możliwość sterowania ilością przeprowadzanych mutacji, tutaj jednak bardzo szybko udało się wysnuć wnioski iż pojedyncza lub maksymalnie podwójna mutacja jest wystarczająca, gdyż większa ich ilość powoduje bardzo szybkie przerzucanie nas w kompletnie inne zbiory rozwiązań.

4.Schemat działania algorytmu:

a) Ustalamy wcześniej wymienione parametry programu.
b) Losujemy populacje wejściową Pw i sortujemy ją rosnąco względem długości ścieżek.
c) Wybieramy z populacji Pw dwa osobniki w następujący sposób: Losujemy **N** osobników, następnie sortujemy je wg wartości funkcji celu. Wybieramy pierwszego z nich. Powtarzamy procedurę aby wybrać drugiego osobnika.
d) Wybrane osobniki krzyżujemy dwukrotnie, tak aby powstały nam dwa osobniki potomne.
e) Powstałe osobniki potomne poddajemy mutacji zgodnie z prawdopodobieństwem P.
f) Dodajemy je do populacji Pn.
g) Jeżeli w populacji Pn znajduje się już L osobników, to idziemy dalej. W przeciwnym wypadku cofamy się do punktu c) i powtarzamy procedurę.
h) Jeżeli w populacji Pn mamy już L osobników, to sumujemy populacje Pn oraz Pw a następnie sortujemy nową populacje względem funkcji celu.
i) Wybieramy L osobników i tworzymy z nich nową populację Pw. Inkrementujemy k. Jeżeli k przewyższy liczbę generacji, to naszym rozwiązaniem jest pierwszy osobnik populacji Pw. Jeżeli k jest mniejsze, to wracamy do punktu c)

5. Przykład praktyczny:

Tabela miast z której skorzystamy:

\	0	1	2	3
0	∞	3	2	1
1	3	∞	5	2
2	3	1	∞	5
3	7	4	2	∞

Parametry wejściowe: L=4, k=1, P=15%, M=1

Wylosowana została następująca populacja wejściowa Pw:

0-1-2-3-0 (koszt=20) **0-1-3-2-0** (koszt=10)

0-2-1-3-0 (koszt=12) **0-3-2-1-0** (koszt=7)

Sortujemy Pw względem funkcji celu, zatem kolejność będzie następująca:

0-3-2-1-0 => **0-1-3-2-0** => **0-2-1-3-0** => **0-1-2-3-0**

Z populacji Pw losujemy teraz dwa osobniki, według przytoczonego wcześniej schematu działania algorytmu.

Wylosowanie osobniki to: P1: **0-3-2-1-0** oraz P2: **0-1-2-3-0**

Następnie trzeba powyższe osobniki skrzyżować, tak aby otrzymać z nich dwóch potomków.

Procedura krzyżowania również była już wcześniej opisana, zatem pominę część losowania indeksów.

Po skrzyżowaniu rodziców P1 oraz P2, otrzymujemy dwójkę potomstwa o następujących ścieżkach: D1: **0-3-1-2-0** oraz D2: **0-3-2-1-0**.

Jak łatwo zauważyć drugi potomek wygląda identycznie jak rodzic, co jednak jest wcale nie rzadkim zjawiskiem kiedy przetwarzamy tak małą ilość miast.

Otrzymanych potomków mutujemy z pewnym prawdopodobieństwem, przyjmijmy na potrzeby przykładu że osobnik pierwszy nie zostanie zmutowany, natomiast drugi tak.

Zatem po zmutowaniu nasz potomek D2 wygląda następująco:

0-1-2-3-0

Otrzymaliśmy zatem dwójkę potomków D1 oraz D2. Dodajemy ich do nowej populacji o nazwie Pn. Sprawdzamy czy populacja Pn liczy już L osobników. Zawiera jedynie dwóch, zatem brakuje jeszcze kolejnych dwóch. Zatem powtarzamy całą procedurę.

Przyjmijmy że w tym miejscu mamy już czwórkę nowych potomków w populacji Pn. Populacja ta wygląda następująco:

0-1-2-3-0

0-3-1-2-0

0-2-1-3-0

0-3-2-1-0

Powyższe osobniki stają się naszą populacją wejściową, sortujemy ją. Ponieważ ustawiliśmy też jedynie jedną generację, to nasz algorytm uważa się za skończony.

Osobnikiem najlepszym jest osobnik znajdujący się na początku listy a zatem:

0-3-2-1-0

Jest to również ścieżka optymalna.

Powyższy przykład traktować jednak należy z pewnym dystansem, ponieważ przyjęta ilość miast jest zbyt mała aby ten algorytm mógł spełniać swoje zadanie. Dlatego też algorytmy metaheurystyczne zalecane są jednak do większych instancji problemów, zwłaszcza że grafy do około 13 miast można policzyć w czasie poniżej 10 sekund przy pomocy algorytmu typu Bruteforce który zawsze da nam optymalne rozwiązanie.

6. Implementacja:

```
public bool CompareLists(List<List<int>> MainList, List<int> list)
```

Powyższa funkcja służy do sprawdzenia, czy w liście *MainList* znajduje się już lista o elementach identycznych jak *list*.

```
private List<List<int>> LoadCities(int NumberOfGenerations)
```

Powyższa funkcja służy do wczytania z pliku wszystkich miast z krawędziami występującymi między nimi, jak również wygenerowaniu listy *L* losowych rozwiązań które posłużą jako pierwsza populacja Pw.

```
public double GetTotalDistance(List<int> order)
```

Jest to funkcja wyliczająca dla danej, pojedynczej ścieżki jej całkowitą odległość.

```
public List<List<int>> Sort(List<List<int>> Lista)
```

Jest to funkcja służąca do sortowania naszej populacji względem funkcji celu (najlepszej ścieżki).

```
public List<List<int>> GenerateRandomPopulation(List<List<int>> Lista,int Size,
int NumberOfCities)
```

Jest to funkcja pozwalająca na wygenerowanie określonej liczby losowych połączeń między daną ilością miast.

```
public double Run(int NumberOfGenerations, int k, int propability, int mutacje,
int tournament)
```

Główna funkcja całego program, odpowiada ona za wywoływanie pozostałych funkcji oraz przyjmuje jako argumenty wszystkie parametry potrzebne do sterowania algorytmem.

7. Badania:

Badania zostały przeprowadzone dla trzech różnych instancji problemu, pobranych ze strony zawierającej rozwiązania optymalne (lub też najlepsze dotychczas znalezione). Instancje te noszą nazwy: *gr17*, *bays29*, *berlin52*. Tych samych miast użyto w poprzednim projekcie, dlatego będzie można łatwo porównać wyniki działań. Testy zostały przeprowadzone dla następujących parametrów algorytmu:

- 50,100,150 osobników w populacji L
- 1000,3000,5000,10000 generacji k
- Prawdopodobieństwo mutacji ustalone na 15%
- Jednorazowo przeprowadzana była jedna mutacja
- 2,3,5 osobników porównywanych z reguły turniejowej

Każdy test powtórzony został 10 razy, następnie wyliczony został średni czas, średnia i mediana ze ścieżek oraz informacja czy znalezione zostało rozwiązanie optymalne (oznaczenie poprzez zieloną kropkę).

Prawdopodobieństwo oraz ilość mutacji zostały przypisane na stałe, ponieważ drogą eksperymentalną udało się dość do wniosku że inne wartości powodują znaczne pogorszenie się otrzymywanych wyników (poprzez zbyt częste zmienianie otoczenia rozwiązań).

Ponadto dołączone zostało porównanie wyników wszystkich trzech wykonanych algorytmów.

Plik gr17

17 miast

Optymalna ścieżka : 2085

Dla tego pliku, każdy test znalazł ścieżkę optymalną.

Dla 2 osobników w turnieju

L \ k	50	100	150
Średni czas[s]			
1000	0,7910	2,4721	4,4732
3000	2,0931	6,8093	15,8249
5000	3,5962	10,5286	26,2375
10000	6,8823	25,4564	42,0184
Średnia długość ścieżki			
1000	2125,6	2118	2144,1
3000	2121,9	2128,3	2094,1
5000	2195,9	2134,8	2092,6
10000	2160,7	2107,1	2120,8
Mediana ze wszystkich ścieżek			
1000	2120	2090	2123
3000	2123	2123	2085
5000	2188	2103	2090
10000	2088	2090	2123
Błąd % (ze średniej)			
1000	1,94	1,58	2,83
3000	1,76	2,07	0,43
5000	5,31	2,38	0,36
10000	3,63	1,05	1,71

Dla 3 osobników w turnieju

L \ k	50	100	150
Średni czas[s]			
1000	0,6860393	2,2881309	4,9242816
3000	2,2041261	6,9043949	12,9167387
5000	3,4931998	11,557661	22,9133106
10000	7,763444	23,6353519	43,3184776
Średnia długość ścieżki			
1000	2167,5	2149,1	2124,8
3000	2130,6	2136,9	2116,5
5000	2177,1	2116	2117,3
10000	2127	2125,8	2114
Mediana ze wszystkich ścieżek			
1000	2170	2153	2090
3000	2090	2103	2103
5000	2188	2090	2103
10000	2090	2123	2103
Błąd % (ze średniej)			
1000	3,956835	3,0743405	1,908872902
3000	2,18705	2,4892086	1,510791367
5000	4,417266	1,4868106	1,549160671
10000	2,014388	1,9568345	1,39088729

Dla 5 osobników w turnieju

L \ k	50	100	150
Średni czas[s]			
1000	0,7740442	2,4231386	5,3713072
3000	2,2311276	6,3923656	14,3718221
5000	3,9452256	10,3225904	23,7533586
10000	7,5954344	20,3571644	43,5434906
Średnia długość ścieżki			
1000	2163,7	2160,1	2110,8
3000	2147,3	2135,7	2108,4
5000	2137,2	2163,5	2139,7
10000	2107,8	2163,8	2121,2
Mediana ze wszystkich ścieżek			
1000	2188	2153	2090
3000	2103	2123	2103
5000	2088	2188	2123
10000	2090	2153	2103
Błąd % (ze średniej)			
1000	3,77458	3,6019185	1,237410072
3000	2,98801	2,4316547	1,122302158
5000	2,503597	3,764988	2,623501199
10000	1,093525	3,7793765	1,736211031

Plik bays29

29 miast

Optymalna ścieżka : 2020

▪ – *znaleziono ścieżkę optymalną*

Dla 2 osobników w turnieju

L \ k	50	100	150
Średni czas[s]			
1000	0,8380479	2,7251558	5,8193328
3000	2,3811362	7,5564322	16,6819542
5000	3,9872281	12,5977205	25,9714854
10000	7,7414427	27,6245801	52,2689896
Średnia długość ścieżki			
1000	2305,8	2263	2159,3
3000	2238,1	2171,3	2167,6
5000	2161,3▪	2250,7	2174,9
10000	2257,2	2191,9▪	2179
Mediana ze wszystkich ścieżek			
1000	2283	2288	2165
3000	2285	2165	2165
5000	2159	2262	2161
10000	2218	2194	2197
Błąd % (ze średniej)			
1000	10,58993	8,5371703	3,563549161
3000	7,342926	4,1390887	3,961630695
5000	3,659472	7,9472422	4,3117506
10000	8,258993	5,1270983	4,508393285

Dla 3 osobników w turnieju

L \ k	50	100	150
Średni czas[s]			
1000	0,8280474	2,6201499	5,3473058
3000	2,3011317	7,7444429	16,4039382
5000	4,0162297	14,4138244	26,1304946
10000	8,2104697	27,2845606	53,2420452
Średnia długość ścieżki			
1000	2358,9	2268,3	2184,1
3000	2278,3	2240,6	2220,6
5000	2274,8	2215,2	2183,4
10000	2288,4	2234	2191,5
Mediana ze wszystkich ścieżek			
1000	2343	2287	2186
3000	2309	2225	2202
5000	2292	2223	2225
10000	2249	2278	2214
Błąd % (ze średniej)			
1000	13,13669	8,7913669	4,752997602
3000	9,270983	7,4628297	6,503597122
5000	9,103118	6,2446043	4,71942446
10000	9,755396	7,146283	5,107913669

Dla 5 osobników w turnieju

L \ k	50	100	150
Średni czas[s]			
1000	0,7760444	2,4901424	5,0622896
3000	2,4901424	7,4064237	14,7658445
5000	4,1992402	11,9316824	25,5714626
10000	7,7394426	26,382509	49,8898535
Średnia długość ścieżki			
1000	2407,5	2201,7	2233,8
3000	2196,6	2268,4	2212,4
5000	2332,2	2193,1	2170,7
10000	2220,7	2255,1	2158,2
Mediana ze wszystkich ścieżek			
1000	2343	2287	2186
3000	2309	2225	2202
5000	2292	2223	2225
10000	2249	2278	2214
Błąd % (ze średniej)			
1000	15,46763	5,5971223	7,136690647
3000	5,352518	8,7961631	6,110311751
5000	11,85612	5,1846523	4,110311751
10000	6,508393	8,1582734	3,510791367

Plik berlin52

52 miasta

Optymalna ścieżka : 7544

▪ – *znaleziono ścieżkę optymalną*

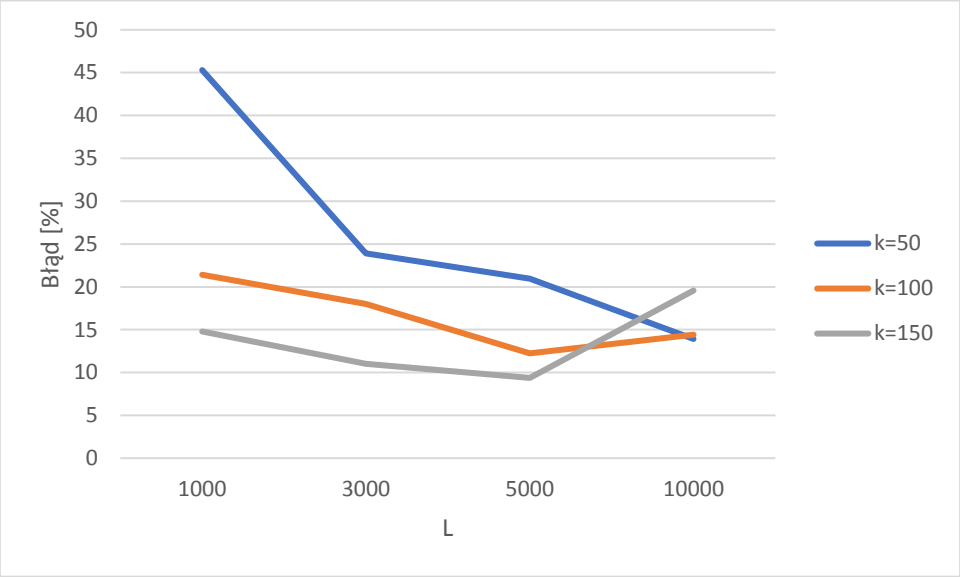
Dla 2 osobników w turnieju

L \ k	50	100	150
Średni czas[s]			
1000	1,4080806	4,4912569	8,4894856
3000	4,0702328	13,8447919	26,6535245
5000	7,7394427	22,8623077	43,8855101
10000	14,16281	53,1570404	92,8843127
Średnia długość ścieżki			
1000	10961	9989,5	9124,7
3000	10119,1	9518,6	9096,1
5000	9893,7	9272,2	9259,7
10000	9944,9	9519,3	9461,8
Mediana ze wszystkich ścieżek			
1000	10961	10152	9086
3000	10357	9573	9118
5000	9877	9382	9219
10000	10225	9529	9541
Błąd % (ze średniej)			
1000	45,29427	32,41649	20,95307529
3000	34,13441	26,174443	20,57396607
5000	31,14661	22,908271	22,74257688
10000	31,82529	26,183722	25,42152704

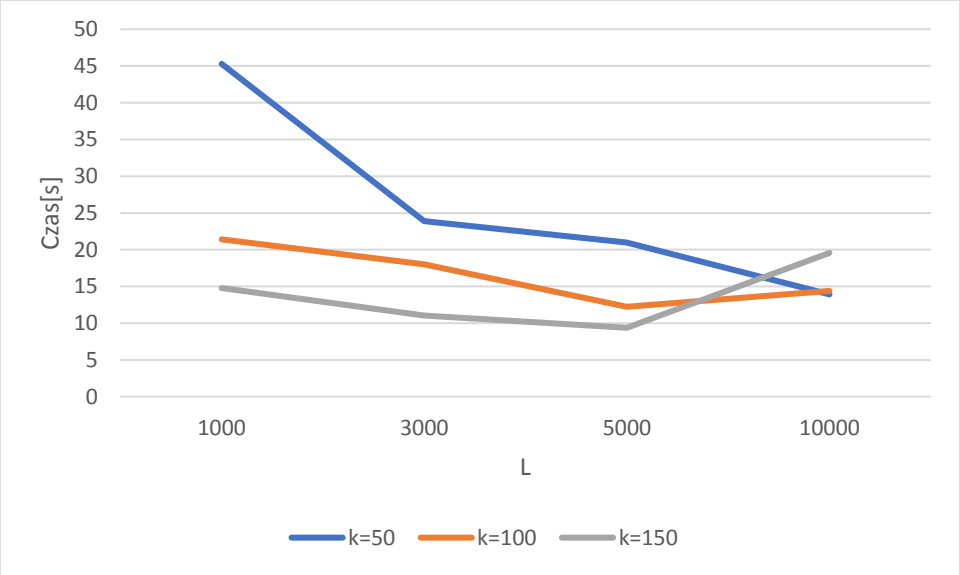
*Można dla tej instancji łatwo zauważyć, że błąd jest znacząco wyższy. Dzieje się tak ponieważ bierzemy średnią ze wszystkich ścieżek, a niektóre z nich bywają bardzo niekorzystne. Gdyby wyliczać błąd na podstawie jedynie najlepszej ścieżki, to wyglądałby on następująco:

Błąd % (ze średniej)			
1000	45,29427	21,407741	14,79321315
3000	23,91304	18,014316	11,02863203
5000	20,97031	12,234889	9,371686108
10000	13,91835	14,408802	19,56521739

Wykres zależności błędu od parametrów



Wykres zależności czasu od parametrów



Porównanie wyników z algorytmem symulowanego wyżarzania oraz podziału i ograniczeń.

Przeprowadzony został test 10 razy dla każdego algorytmu, wybierając na podstawie przeprowadzonych wcześniej badań najlepsze możliwe parametry. W tabeli przedstawione są najlepsze wyniki jakie udało się uzyskać.

Dla wyżarzania:

Funkcja obniżania temperatury : $F_3 = 0.99999 * T_i$

Temperatura początkowa : $10 * 10^{10}$

Dla genetycznego:

Zależnie od pliku. Dla małych instancji wystarczały mniejsze rozmiary populacji/generacji, aby zaoszczędzić czas. Najlepsze, optymalne dla większości instancji miast parametry algorytmu to: L=6500, k=130, P=15%, pojedyncza mutacja metodą inwersji oraz po dwa osobniki w każdym turnieju.

	Rodzaj algorytmu					
Plik	**Branch&Bound**		**Wyżarzanie**		**Genetyczny**	
	Wynik	**Czas[s]**	**Wynik**	**Czas[s]**	**Wynik**	**Czas[s]**
Gr17	2085	921	2085	1.831	2085	2,173
Fri26	-	-	937	1.295	937	38,125
Bays29	-	-	2020	2.497	2020	3,461
Dantzig42	-	-	755	3.812	820	48,060
Berlin52	-	-	7837	4.772	8251	58,522

Optymalne ścieżki: **Fri26(937),Gr17(2085),Bays29(2020),Berlin52(7544),Dantzig42(699)**

Wnioski

Jak można zauważyć w powyższej tabeli, wydajnościowo najlepiej sprawdza się algorytm symulowanego wyżarzania. Dzieje się tak ze względu na prostotę implementacji tego algorytmu i nikłą ilość parametrów którą można sterować, przez co dużo łatwiej jest dobrać optymalne parametry dla każdej instancji. Natomiast w przypadku algorytmu genetycznego, parametrów ilość parametrów jest znacząca przez co dla każdej instancji najlepsze parametry mogą być znacząco inne. Dlatego też najlepszym algorytmem wydaje się wyżarzanie, ze względu na jego elastyczność względem różnych instancji, bardzo krótkiego czasu działania oraz całkiem zadowalających wyników. Algorytmu branch&bound nie ma sposobu przyrównać do dwóch pozostałych, ze względu na jego złożoność

czasową, która dla pesymistycznego przypadku pokrywa się ze złożonością algorytmu typu BruteForce. Jednak jego dużym plusem jest pewność otrzymania wyniku optymalnego, dlatego też dla instancji o niewielkich rozmiarach jest to zalecane rozwiązanie. Natomiast dla pozostałych lepiej zastosować symulowane wyżarzanie.