Porównanie algorytmów rozwiązujących problem komiwojażera

dokumentacja do projektu z przedmiotu „Teoria Obliczeń i Złożoności Obliczeniowej”

Marcin Rociek
Informatyka, III rok

28 grudnia 2000

Spis treści

1 Wstęp

Komiwojażer chce odwiedzić pewną liczbę miast, przebywając w każdym z nich tylko raz i wrócić do punktu startu, ale w taki sposób aby przebyć w sumie możliwie najkrótszą trasę. Z algorytmicznego punktu widzenia sprowadza się to do znalezienia w grafie *n*-wierzchołkowym (skierowanym bądź nieskierowanym) cyklu o najmniejszej wadze (np. długości). Każda permutacja *n* miast jest jakimś rozwiązaniem (pełnym objazdem *n* miast), wystarczy więc spośród wszystkich *n*! permutacji wybrać tę o najmniejszej wadze, która to będzie optymalną trasą komiwojażera. Cały problem polega jednak na tym, że *n*! to bardzo dużo: zakładając (przykładowo), że dla 30 miast jakiś komputer potrafi w ciągu sekundy sprawdzić miliard permutacji (różnych rozwiązań), to na optymalne rozwiązanie będzie trzeba czekać ponad 8 biliardów lat (ok. $8,4 \times 10^{15}$).

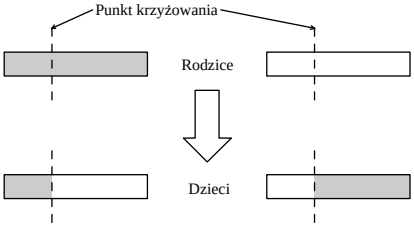
Powstało kilka algorytmów, które w relatywnie krótkim czasie wyznaczają rozwiązanie bliskie optymalnemu. Między innymi są to algorytmy **genetyczne**, oraz algorytmy oparte na symulacji **kolonii mrówek**. W projekcie zostały zaimplementowane i porównane algorytmy: genetyczny z selekcją CHC i reinicjalizacją, oraz system kolonii mrówek $\mathcal{MM}$AS . Nadają się one jak najbardziej do tego projektu, ponieważ mogą rozwiązywać problem komiwojażera dla dowolnego grafu (nie tylko dla szczególnego przypadku, w którym wierzchołki umieszczone są na płaszczyźnie, a wagi są odległościami).

2 Algorytm genetyczny

2.1 Opis algorytmu

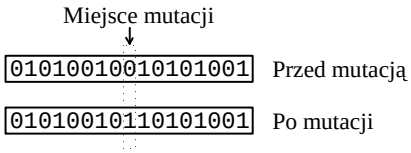
Algorytmy genetyczne są algorytmami wyszukującymi najlepsze rozwiązanie danego problemu korzystając z mechanizmów selekcji naturalnej oraz genetyki. Wykorzystywane są do rozwiązywania wielkich problemów, w przypadku których zawodzą tradycyjne metody (takich jak na przykład problem komiwojażera). Algorytm genetyczny pracuje na populacji osobników (chromosomów), z których każdy reprezentuje jakieś możliwe rozwiązanie danego problemu. Każdy z chromosomów ma przypisaną wartość dopasowania, wyznaczoną przez specyficzną dla zadanego problemu funkcję. Osobniki, których wartość dopasowania jest lepsza niż u innych, mają większe szanse na reprodukcję, a ich potomkowie dziedziczą cechy po rodzicach.

Algorytmy genetyczne generalnie używają trzech podstawowych operatorów: selekcji, krzyżowania i mutacji. W każdej iteracji t generuje się populację osobników $P(t)$, reprezentowanych przez ciągi bitów o jednakowej długości. Każdy z osobników przedstawia pewne rozwiązanie danego zadania. Każde rozwiązanie ocenia się na podstawie pewnej miary jego dopasowania. Nową populację (w iteracji $t+1$) tworzy się poprzez selekcję wybranych osobników z iteracji t. W klasycznym algorytmie genetycznym prawdopodobieństwo wybrania danego osobnika do następnego pokolenia jest proporcjonalne do wartości dopasowania: lepiej dopasowane chromosomy z większym prawdopodobieństwem „przeżyją" do następnego pokolenia. Pewne osobniki nowej populacji podlegają dodatkowo transformacjom za pomocą operatorów mutacji i krzyżowania. Klasyczny operator krzyżowania wybiera losowo dwa chromosomy oraz miejsce krzyżowania, następnie rozcina w tym miejscu oba chromosomy i zamienia miejscami odpowiednie części chromosomów (rys. 1). W wyniku tej operacji tworzone są dwa nowe chromosomy - potomkowie.



Rysunek 1: Klasyczny operator krzyżowania

Klasyczny operator mutacji zamienia wartość losowo wybranego bitu w ciągu bitów reprezentującym chromosom (rys. 2), pozwala to m.in. uniknąć zablokowania się algorytmu w lokalnym minimum.



Rysunek 2: Klasyczny operator mutacji

2.2 Modyfikacje algorytmu specyficzne dla problemu komiwojażera

W przypadku zastosowania algorytmu genetycznego do rozwiązania problemu komiwojażera, wartością dopasowania chromosomu będzie długość (koszt) uzyskanej trasy. W przypadku gdy dane są wagi poszczególnych krawędzi grafu, będzie to po prostu suma wag krawędzi wchodzących w skład trasy. Gdy dane są współrzędne wierzchołków (np. na płaszczyźnie), wartością dopasowania będzie suma odległości między każdą parą wierzchołków wchodzących w skład trasy:

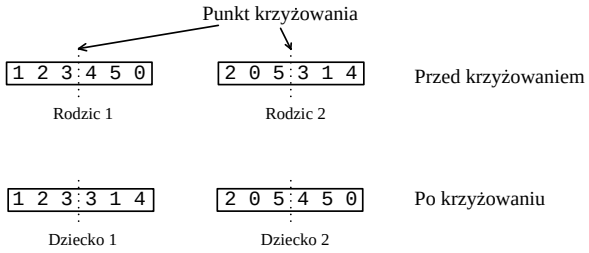
$$F = \sum_{i=1}^{N} \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}$$

Reprezentacja

Dane rozwiązanie problemu komiwojażera (chromosom) nie jest reprezentowane przez ciąg bitów ale przez ciąg liczb, określających numery kolejno odwiedzanych wierzchołków. Np. zakładając, że graf składa się z 5 wierzchołków: 0, 1, 2, 3, 4, możliwą trasą reprezentowaną przez chromosom może być np. ciąg liczb 3 1 4 0 2, oznaczający trasę komiwojażera rozpoczynającą się w wierzchołku 3, przechodzącą przez wierzchołki 1, 4, 0, 2 a następnie powracającą do punktu wyjścia, czyli wierzchołka 3. Gdy graf ma N wierzchołków, populacja inicjalizowana jest w ten sposób, że do każdego z chromosomów (każdy o długości N) wstawiane są losowo liczby od 0 do $N-1$ tak, aby każda liczba powtarzała się tylko raz. Tak skonstruowane chromosomy reprezentują dopuszczalne trasy komiwojażera.

Operator krzyżowania

Z powodu przyjętej reprezentacji chromosomów, klasyczny operator krzyżowania nie nadaje się do problemu komiwojażera. Rysunek 3 pokazuje jak klasyczny operator krzyżowania prowadzi do powstawania potomków reprezentujących nieprawidłowe trasy. Potomek 1 jest nieprawidłowy, ponieważ wierzchołki 1 i 3 pojawiają się w nim dwukrotnie, natomiast wierzchołki 0 i 5 nie pojawiają się w ogóle. Potomek 2 również jest nieprawidłowy, ponieważ wierzchołki 0 i 5 pojawiają się w nim dwukrotnie, zaś wierzchołki 1 i 3 się nie pojawiają.



Rysunek 3: Klasyczny operator krzyżowania nie nadaje się do problemu komiwojażera, w którym każdy z osobników jest listą kolejno odwiedzanych miast

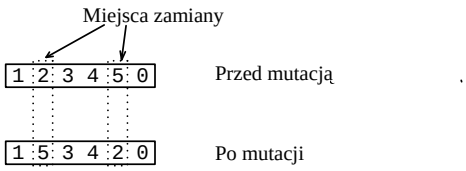
Istnieje wiele różnych operatorów krzyżowania, jednym z nich jest tzw. *krzyżowanie zachłanne* wprowadzone w 1985 r. przez Grafenstette'a. Polega ono na tym, że jako pierwszy wierzchołek tworzonego potomka wybierany jest pierwszy wierzchołek z jednego z rodziców. Następnie porównuje się wagi krawędzi wychodzących z tego wierzchołka w obu rodzicach i do potomka wstawia ten wierzchołek, który leży bliżej. Jeśli jeden z tych wierzchołków już występuje w potomku, wybierany jest ten drugi. Jeśli oba wierzchołki występują już w potomku, wybierany jest losowo jakiś wierzchołek, który jeszcze w potomku nie wystąpił.

Przykładowo mamy dwóch rodziców 1 2 3 4 5 0 i 4 1 3 2 0 5. Wybieramy pierwszy wierzchołek z (przykładowo) drugiego rodzica jako początkowy wierzchołek potomka: 4 x x x x x. Następnie sprawdzane są krawędzie w obu rodzicach, wychodzące z wierzchołka 4, czyli (4, 5) i (4, 1). Jeśli krawędź (4, 1) jest krótsza, do potomka wstawiany jest wierzchołek 1: 4 1 x x x x. Następnie sprawdzane są krawędzie wychodzące z 1, czyli (1, 2) i (1, 3); jeśli krótsza jest krawędź (1, 2), do potomka wstawiany jest wierzchołek 2: 4 1 2 x x x. Na podobnej zasadzie sprawdzając krawędzie (2, 3) i (2, 0) wprowadzamy do potomka wierzchołek 0: 4 1 2 0 x x. Krawędzie wychodzące z 0 to (0, 1) i (0, 5); ponieważ 1 już jest w potomku, wprowadzony do niego zostaje siłą rzeczy wierzchołek 5: 4 1 2 0 5 x. Wierzchołki leżące za wierzchołkiem 5 to 0 i 4, ale ponieważ oba z nich znajdują się już w potomku, wybrany zostaje jeszcze nie wybrany wierzchołek 3, doprowadzając do zakończenia tworzenia potomka: 4 1 2 0 5 3. Na tej samej zasadzie można stworzyć drugiego potomka (zaczynając od wybrania pierwszego wierzchołka z pierwszego rodzica): 1 2 0 5 4 3. Po tak przeprowadzonej operacji krzyżowania oba potomki reprezentują poprawne trasy komiwojażera.

Operator mutacji

Klasycznego operatora mutacji nie można użyć z tego samego powodu, dla jakiego przy przyjętej reprezentacji tras komiwojażera nie mogliśmy użyć klasycznego operatora krzyżowania. Zakładając np., że mamy chromosom 1 2 3 4 5 0 wybieramy losowo miejsce mutacji i zamieniamy 3 na 5, doprowadzając do powstania nieprawidłowego chromosomu 1 2 5 4 5 0, w którym wierzchołek 5 pojawia się dwukrotnie, natomiast wierzchołek 3 nie pojawia się wcale.

Rysunek 4 przedstawia stosowany operator mutacji: wybierane są losowo dwa wierzchołki w chromosomie, których wartości są zamieniane. W ten sposób po operacji mutacji wciąż mamy chromosom reprezentujący prawidłową trasę komiwojażera.



Rysunek 4: Mutacja polegająca na zamianie dwóch losowo wybranych wierzchołków

Operator selekcji

W przypadku tradycyjnego operatora selekcji (ruletki) najlepsze osobniki mają największą szansę na przeżycie, ale niekoniecznie przeżywają. Zamiast tego można użyć selekcji CHC (opisanej przez Eshelmana w r. 1991). W przypadku populacji N osobników, wykorzystując tradycyjną selekcję tworzy się N potomków, których dołącza się do populacji tworząc grupę $2N$ osobników. Następnie z tej puli wybiera się N najlepszych (pod względem wartości dopasowania), którzy przeżywają do następnego pokolenia. Rozwiązanie takie prowadzi do szybszego zbiegania się wyników do pewnej wartości. Szybsze zbieganie się może jednak znaczyć mniejszą eksplorację grafu. Aby tego uniknąć można stosować różne metody. W pracy [1] opisane jest rozwiązanie, w którym po osiągnięciu pewnej wartości (zbiegnięciu się wszystkich rozwiązań) 10% populacji (sami najlepsi osobnicy) pozostają przy życiu, natomiast reszta populacji jest losowo reinicjalizowana. Innym możliwym rozwiązaniem ([2]) jest reinicjalizacja części populacji (bądź nawet całej populacji za wyjątkiem jednego najlepszego osobnika) gdy tylko pewne n najlepszych osobników (elita) posiada tą samą wartość dopasowania (koszt trasy).

2.3 Złożoność obliczeniowa

Algorytm genetyczny można zapisać w następującej postaci:

```
1  Inicjalizacja
2  Pętla główna
  2.1  Krzyżowanie chromosomów
  2.2  Mutacje chromosomów
  2.3  Sortowanie chromosomów w populacji
  2.4  Wyznaczenie "elity"
  2.5  Ewentualna reinicjalizacja części populacji
```

Przyjmując następujące oznaczenia: i - ilość iteracji algorytmu, N - ilość chromosomów, n - długość chromosomu (ilość miast), p - ilość chromosomów nie podlegających reinicjalizacji; można stwierdzić, że:

- Krok 1 ma złożoność $O(N \cdot n)$ - każdy z N chromosomów w populacji jest inicjalizowany *jakimś* rozwiązaniem. W tej konkretnej implementacji jest to rozwiązane przez wypełnienie chromosomu kolejnymi liczbami od 0 do $n-1$, a następnie n-krotną mutację. Złożoność inicjalizacji ma jednak bardzo mały wpływ na złożoność całego algorytmu (inicjalizacja wykonywana jest tylko raz).

- Krok 2 jest wykonywany i razy, a więc bezpośrednio od wartości i zależy czas trwania algorytmu. Ilość iteracji można ustalić odgórnie, najlepiej jednak dostosować ją indywidualnie do konkretnego problemu. Jednym ze sposobów jest ustalenie ilości iteracji poprzez zadanie górnego limitu czasu jaki ma być wykonywany program, bądź też kosztu trasy jaki jest dla nas zadowalający, po osiągnięciu którego program ma przestać działać (oczywiście istnieje wtedy niebezpieczeństwo nie otrzymania rozwiązania nigdy - np w przypadku podania kosztu trasy mniejszego od minimalnego).

- Złożoność kroku 2.1 zależy od tego, w jaki sposób dokonujemy wyboru rodziców, możemy bowiem dokonać wyboru zupełnie losowego (nie patrząc na koszty tras reprezentowanych przez potencjalnych rodziców), bądź też wybierać ich metodą *ruletki* tj. prawdopodobieństwo wybrania jakiegoś chromosomu na rodzica jest odwrotnie proporcjonalne do jego wartości dopasowania (kosztu trasy) - im koszt

mniejszy, tym prawdopodobieństwo większe. W przypadku wyboru losowego krok 2.1 ma złożoność $O(N \cdot n^2)$ (tworzonych jest N potomków, w każdym z nich trzeba wyznaczyć n wierzchołków, kolejne n wprowadza operacja szukania wolnego wierzchołka do wprowadzenia (jeden z etapów krzyżowania)). W przypadku selekcji metodą ruletki złożoność wynosi $O(N^2 \cdot n^2)$, kolejne N wprowadza operacja szukania odpowiedniego kandydata na rodzica, stosownie do wartościami dopasowania.

- Krok 2.2 ma złożoność $O(N \cdot n)$ z małym współczynnikiem - prawdopodobieństwo mutacji w tej konkretnej implementacji wynosi 0.047, a więc taka część z N chromosomów podlega mutacji, dodatkowo po przeprowadzeniu mutacji konieczne jest wyznaczenie od nowa kosztu chromosomu (czyli n-krotne dodawanie kosztów poszczególnych krawędzi).

- Krok 2.3 ma złożoność $O(N)$, a to za sprawą zastosowanego algorytmu sortującego, tzw. *radix*: z 32-bitowego klucza względem którego przeprowadzane jest sortowanie (czyli z kosztu trasy) wydzielana jest czterokrotnie 8-bitowa część, będąca indeksem w 256-elementowej tablicy haszującej. Po każdym umieszczeniu wszystkich elementów w tablicy następuje ich złożenie w nowej kolejności. Ponieważ chromosomy powiązane są w listę, większość operacji to tylko operacje na wskaźnikach.

- Krok 2.4 ma również złożoność $O(N)$ i to w najgorszym przypadku, w którym wszystkie chromosomy mają takie same wartości dopasowania (koszty trasy) i nie następuje reinicjalizacja (bo np. jest wyłączona). Wyznaczanie „elity” sprowadza się bowiem do wyznaczenia ilości najlepszych chromosomów, które mają takie same wartości, a więc w zasadzie wpływ tego kroku na złożoność całego algorytmu jest nikły.

- Krok 2.5 (wykonywany tylko jeśli spełniony jest odpowiedni warunek, zależny od wielkości „elity” wyznaczonej w kroku 2.4) ma złożoność $O(N \cdot n)$ - każdy z $N - p$ chromosomów w populacji (pewna ilość chromosomów p musi pozostać nietknięta) przechodzi n-krotną mutację (tworzone jest jakieś nowe losowe rozwiązanie niezależne zupełnie od poprzedniej wartości chromosomu). Następnie populacja musi znów zostać przesortowana ($O(N)$).

Pomijając inicjalizację, algorytm ma więc złożoność $O(i \cdot N^2 \cdot n^2)$ - i-krotnie wykonywana jest iteracja o złożoności $O(N^2 \cdot n^2)$. Kwestia wartości parametrów i i N jest kwestią umowną (bądź co bądź jedynie wartość n jest dana odgórnie: ilość miast jest zależna od konkretnego problemu). Zwiększając i zwiększamy ilość iteracji, natomiast zwiększając N zwiększamy wielkość populacji chromosomów. Wyniki eksperymentalne pokazują jak stosunek tych wartości wpływa na skuteczność działania algorytmu.

2.4 Wyniki eksperymentalne

Testy zostały w większości przypadków przeprowadzane na danych z TSPLIB - jest to zbiór problemów komiwojażera o różnej wielkości, dla których często znane jest rozwiązanie minimalne, co pozwala m.in. określić skuteczność jakiegoś algorytmu.

2.4.1 Wpływ reinicjalizacji

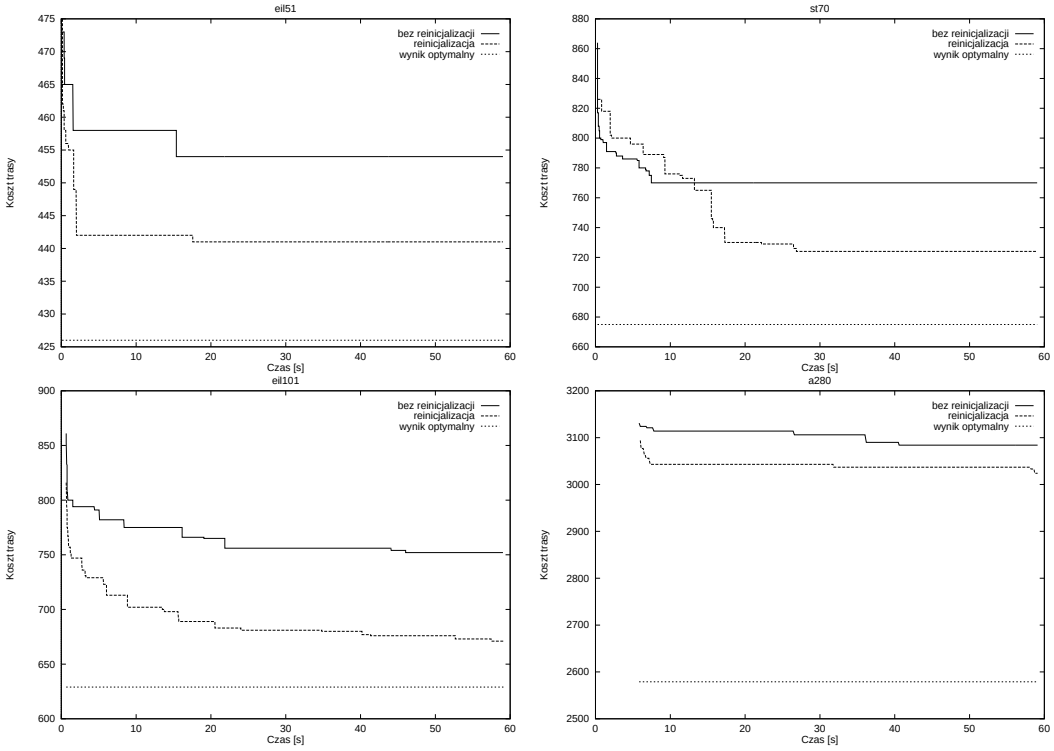
Wykresy przedstawione na rysunku 5 pokazują zachowanie się algorytmu z włączoną i wyłączoną reinicjalizacją. Jak widać, mimo iż sama reinicjalizacja kosztuje (trzeba stworzyć nowe losowe rozwiązania) opłaca się jej stosowanie - znalezione w tym samym czasie trasy są zwykle krótsze niż w przypadku algorytmów stosujących tylko krzyżowanie, mutację i selekcję.

2.4.2 Wpływ rodzaju krzyżowania

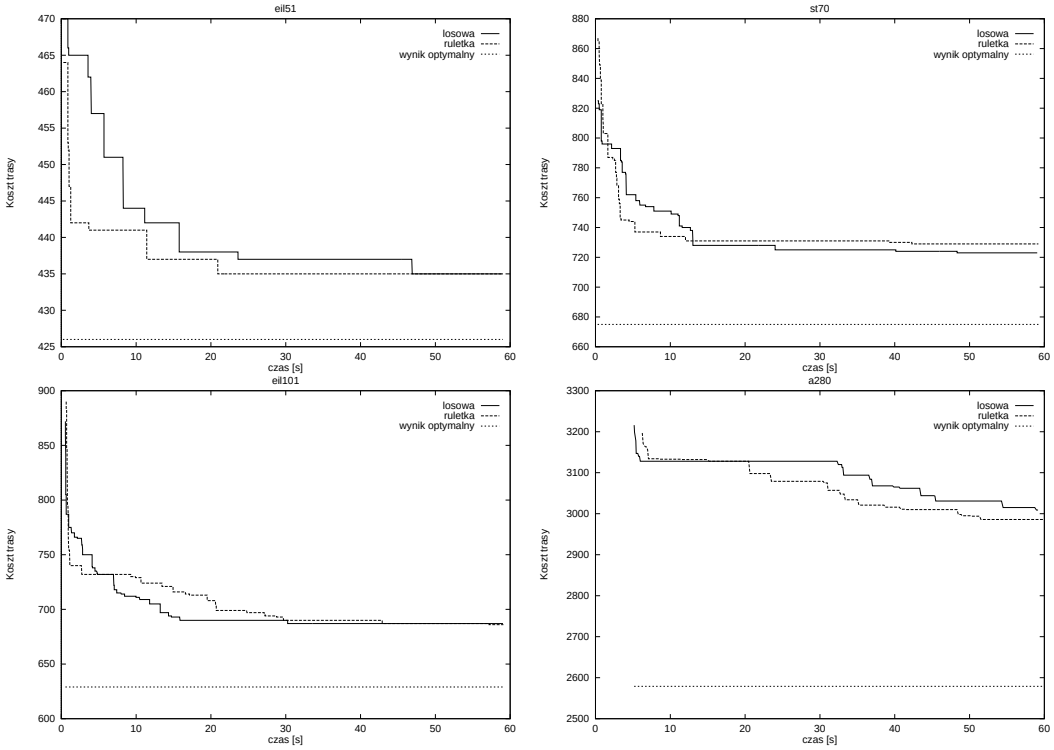
Rysunek 6 przedstawia zachowanie się algorytmu dla różnych metod doboru rodziców w procesie krzyżowania. Jak widać opłacalność wyboru bardziej kosztownej metody ruletki jest problematyczna - choć być może jest to efekt „kulawej” implementacji.

2.4.3 Wielkość populacji

W tablicy 1 zestawiono przykładowe wyniki działania programu dla różnych wielkości populacji (od 30% do 200% ilości wierzchołków), przy jednakowym limicie czasowym (30 s). Widać, że optymalna wielkość populacji silnie zależy od konkretnego problemu.



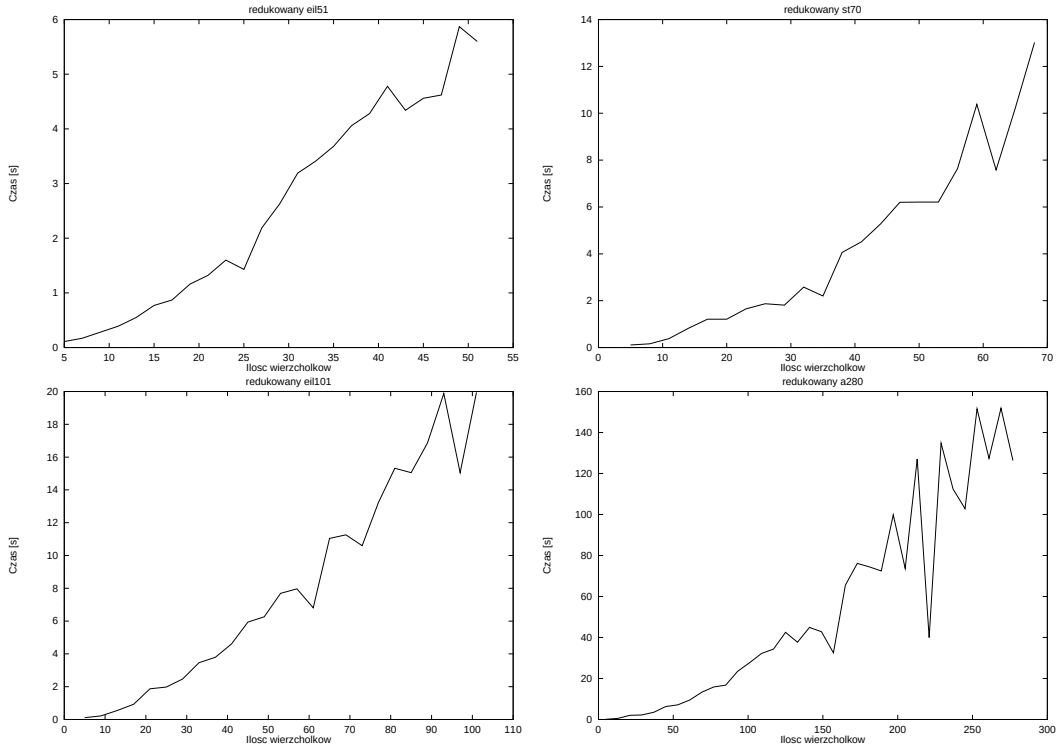
Rysunek 5: Wpływ reinicjalizacji na koszt w miarę działania algorytmu (dla zwiększenia czytelności pominięto pierwsze 30 iteracji).



Rysunek 6: Wpływ rodzaju krzyżowania na koszt w miarę działania algorytmu (dla zwiększenia czytelności pominięto pierwsze 30 iteracji).

2.4.4 Złożoność

Testowanie złożoności wykonane zostało również w oparciu te same problemy z TSPLIB, tj. eil51, st70, eil101 i a280. Z tym, że problemy te były modyfikowane, w taki sposób że wycinana była z nich pewna ilość wierzchołków. Oczywiście zmieniało to wartość minimalnej trasy, ale nie to było istotne, a czas działania programu w zależności od ilości wierzchołków, przy domyślnym ustawieniu parametrów. Wyniki zaprezentowane są na rysunku 7. Z wykresów wynika, że faktyczna złożoność jest nieco lepsza niż $O(n^2)$.

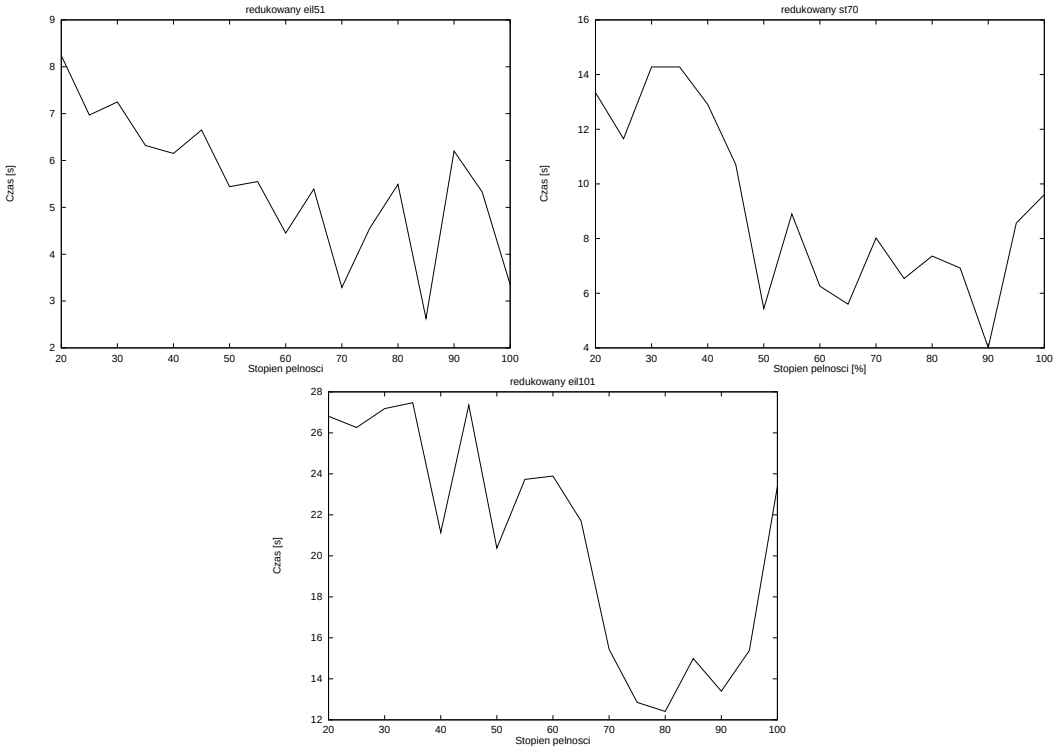


Rysunek 7: Zależność czasu wykonywania się programu od ilości wierzchołków (przy domyślnym ustawieniu parametrów).

Wszystkie dotychczasowe testy przeprowadzane były na grafach pełnych - takie bowiem są grafy z TSPLIB. Rysunek 8 przedstawia zależności czasu wykonywania się programu od ilości krawędzi w grafie. Do tego celu znów posłużyły te same grafy z TSPLIB, tym razem z wycinanymi krawędziami (co oczywiście również wpływa na wartość minimalnej trasy). Widać, że w przypadku algorytmu genetycznego, czas znalezienia rozwiązania maleje wraz z wrostem współczynnika pełności grafu.

	eil51 (426)		eil101 (629)		a280 (2579)	
populacja [%]	iteracje	wynik	iteracje	wynik	iteracje	wynik
30	20547	460	2370	701	261	3237
40	6024	**434**	1606	742	193	3243
50	4847	446	1245	737	200	3174
60	3938	455	1056	764	193	3077
70	3301	455	953	725	135	3088
80	2958	443	766	692	136	3147
90	2751	445	690	749	193	3129
100	2320	438	761	697	109	3141
110	2200	445	697	692	187	3027
120	2078	463	559	727	108	3122
130	1809	454	556	688	111	3009
140	1712	444	507	736	88	3067
150	1627	445	454	702	106	2997
160	1462	443	522	692	148	3032
170	1406	440	429	696	68	3072
180	1315	455	429	**679**	143	3064
190	1304	438	411	683	108	3050
200	1199	457	382	694	121	**2993**

Tablica 1: Wpływ wielkości populacji na wynik. Liczby w nawiasach oznaczają optymalny wynik danego problemu. Liczby pogrubione oznaczają najlepszy uzyskany rezultat.



Rysunek 8: Zależność czasu wykonywania się programu od pełności grafu (przy domyślnym ustawieniu parametrów).

3 System kolonii mrówek $\mathcal{MM}$AS ($\mathcal{MAX}$-$\mathcal{MIN}$Ant System)

3.1 Opis algorytmu

Działanie algorytmów wykorzystujących symulację kolonii mrówek wzorowane jest na zachowaniu prawdziwych mrówek. Prawdziwe mrówki są zdolne do znalezienia najkrótszej ścieżki łączącej mrowisko ze źródłem pokarmu bez użycia bodźców wizualnych. Ponadto są zdolne do przystosowywania się do zmian otoczenia, np. do znalezienia nowej najkrótszej ścieżki w przypadku, gdy stara nie jest już „przejezdna" (np. znalazła się na niej jakaś przeszkoda). Podstawowym środkiem umożliwiającym mrówkom uformowanie i utrzymanie takiej trasy są ścieżki feromonowe. Mrówki zostawiają pewną ilość feromonu podczas chodzenia, każda z mrówek zaś probabilistycznie decyduje się na wybranie trasy bogatej w feromon. To elementarne zachowanie pozwala wyjaśnić w jaki sposób mrówki znajdują najkrótszą ścieżkę.

3.1.1 Algorytmy ACO

W algorytmach ACO (*Ant Colony Optimization*) mrówkami są prości agenci konstruujący trasy przechodząc z wierzchołka do wierzchołka w danym grafie. Mrówki konstruują rozwiązania kierując się ścieżkami feromonowymi oraz z góry zadanymi wartościami heurystycznymi (np. kosztami krawędzi w grafie). W przypadku rozwiązywania problemu komiwojażera natężenie feromonu $r_{ij}(t)$ jest przypisane do każdej krawędzi (i, j), gdzie $r_{ij}(t)$ jest numeryczną informacją modyfikowaną w czasie trwania algorytmu, zaś t jest numerem iteracji. W przypadku grafów nieskierowanych, zawsze $r_{ij}(t) = r_{ji}(t)$.

Na początku każda z m mrówek umieszczana jest w losowo wybranym wierzchołku grafu. Mrówka konstruuje trasę w następujący sposób: przebywając w wierzchołku i wybiera jedno z wciąż nie odwiedzonych miast j probabilistycznie, bazując na natężeniu feromonu $r_{ij}(t)$ na krawędzi między wierzchołkiem i a j oraz na informacji heurystycznej, czyli w tym przypadku na koszcie krawędzi. Mrówka preferuje wierzchołki znajdujące się blisko, które są połączone krawędziami o dużym natężeniu feromonu. Aby skonstruować prawidłowe rozwiązanie mrówka posiada pewną pamięć, w której przechowuje listę dotychczas odwiedzonych wierzchołków. Pamięć ta jest używana do określenia, które wierzchołki mogą być jeszcze odwiedzone, gwarantując stworzenie poprawnego rozwiązania.

Kiedy wszystkie mrówki skonstruują swoje trasy, następuje uaktualnienie natężenia feromonu na krawędziach. Zwykle sprowadza się to do zmniejszenia wartości feromonów o stały współczynnik, a następnie umożliwienia mrówkom dodania feromonów do krawędzi przez nie odwiedzonych. Uaktualnienie natężenia feromonu jest dokonywane w ten sposób, że krawędzie zawarte w najkrótszych trasach i/lub odwiedzane przez dużą ilość mrówek otrzymują większą dawkę feromonów i w ten sposób są później częściej wybierane w kolejnych iteracjach algorytmu. W tym znaczeniu wielkość feromonu r_{ij} reprezentuje wyuczone żądanie wybrania wierzchołka j, gdy mrówka przebywa w wierzchołku i.

Istnieje wiele odmian algorytmów wykorzystujących powyższą zasadę działania. Różnią się one między sobą sposobem wyboru następnego wierzchołka j, sposobem uaktualniania ścieżek feromonowych (np. mogą tego dokonywać wszystkie mrówki, najlepsza mrówka w danej iteracji, najlepsza mrówka w ogóle itp.). Poniżej omówiony algorytm $\mathcal{MM}$AS posiada ponadto ograniczenia na dopuszczalne wartości feromonów na krawędziach.

3.1.2 Algorytm $\mathcal{MM}$AS

Konstrukcja trasy

Na wstępie każda mrówka umieszczona jest w dowolnie wybranym wierzchołku. W każdym kolejnym kroku mrówka wybiera kolejny wierzchołek, do którego ma się udać. Prawdopodobieństwo, że w t-ej iteracji mrówka k, znajdująca się w wierzchołku i, zdecyduje się udać do wierzchołka j, jest określone następujaym wzorem:

$$p_{ij}^k(t) = \frac{[r_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in \mathcal{N}_i^k} [r_{il}(t)]^\alpha \cdot [\eta_{il}]^\beta} \qquad \text{jeśli } j \in \mathcal{N}_i^k$$

gdzie $\eta_{ij} = 1/d_{ij}$ jest z góry zadaną wartością heurystyczną (d_{ij} to koszt krawędzi pomiędzy wierzchołkiem i a j), α i β są dwoma parametrami określającymi wpływ ścieżek feromonowych oraz informacji heurystycznych, zaś $\mathcal{N}_i^k$ jest zbiorem miast, których mrówka k jeszcze nie odwiedziła, a które może odwiedzić. Rola parametrów α i β jest następująca: jeśli $\alpha = 0$, wpływ na decyzję mrówki mają tylko infomacje o kosztach krawędzi, najprawdopodobniejsze jest więc wybranie najbliższych wierzchołków (jak w klasycznym stochastycznym algorytmie zachłannym). Jeśli $\beta = 0$, wykorzystywana jest zaś tylko informacja feromonowa (prowadzi to do sytuacji *stagnacji*, w której wszystkie mrówki podążają tą samą ścieżką, tworząc to samo rozwiązanie).

Uaktualnianie ścieżek feromonowych

Gdy wszystkie mrówki skonstruowały swoje trasy, ścieżki feromonowe są uaktualniane. Dokonuje się tego najpierw poprzez zmniejszenie wartości feromonu na *wszystkich* krawędziach o stały współczynnik, a następnie dopuszczając tylko najlepszą mrówkę do dodania do krawędzi należących do wytyczonej przez nią trasy pewnej wartości feromonu:

$$r_{ij}(t+1) = \rho \cdot r_{ij}(t) + \Delta r_{ij}^{best}$$

gdzie $\Delta r_{ij}^{best} = 1/L^{best}$ (L^{best} - długość najlepszej trasy). Mrówką, która jest dopuszczona do uaktualnienia zawartości feromonu może być mrówka najlepsza w danej iteracji T^{ib}, bądź mrówka najlepsza w ogóle T^{gb}. W następstwie tego, jeśli pewne krawędzie są często wykorzystywane w najlepszych trasach, one właśnie otrzymają dużą dawkę feromonu.

Limity ścieżek feromonowych

W algorytmie $\mathcal{MM}$AS zakres dopuszczalnych wartości feromonu na wszystkich ścieżkach jest ograniczony do przedziału $[r_{min}, r_{max}]$, tzn. $\forall r_{ij} : r_{min} \leq r_{ij} \leq r_{max}$. Ograniczenie to jest wprowadzone w celu uniknięcia stagnacji. W szczególności, wprowadzenie ograniczeń wartości feromonów bezpośrednio wpływa na ograniczenie prawdopodobieństwa p_{ij} wybrania przez mrówkę znajdującą się w wierzchołku i wierzchołka j do przedziału $[p_{min}, p_{max}]$, gdzie $0 \leq p_{min} \leq p_{ij} \leq p_{max} \leq 1$. Tylko w przypadku, gdy mrówka ma tylko jedno miasto do wyboru, wtedy $p_{min} = p_{max} = 1$. Większe znaczenie mają dolne ograniczenia wartości feromonu, ponieważ maksymalna ilość feromonu i tak jest ograniczana przez parowanie.

Jedną z metod wyznaczania limitów natężenia feromonów jest metoda szczegółowo omówiona w [4], w której limity wyznaczane są z następujących wzorów:

$$r_{max} = \frac{1}{1-\rho} \cdot \frac{1}{f(s^{opt})}$$

$$r_{min} = \frac{r_{max} \cdot (1 - \sqrt[n]{p_{best}})}{(avg-1) \cdot \sqrt[n]{p_{best}}}$$

gdzie $f(s^{opt})$ jest kosztem optymalnej trasy, p_{best} prawdopodobieństwem wybrania najlepszego rozwiązania po zbiegnięciu się algorytmu, zaś $avg = n/2$ jest średnią ilością miast spośród których mrówka musi wybierać w każdym kroku konstruowania trasy.

Inicjalizacja ścieżek feromonowych

W algorytmie $\mathcal{MM}$AS ścieżki feromonowe są inicjalizowane wartością r_{max} tj. górnym limitem dopuszczalnych wartości. Po takiej inicjalizacji eksploracja tras na początku działania algorytmu jest zwiększona, ponieważ względne różnice między wartościami feromonów w poszczególnych ścieżkach są mniej widoczne.

3.2 Złożoność obliczeniowa

Algorytm $\mathcal{MM}$AS można zapisać w następującej postaci:

```
1  Inicjalizacja
2  Pętla główna
  2.1  Tworzenie przez mrówki tras
  2.2  Parowanie feromonu
  2.3  Uaktualnienie feromonu przez mrówki
```

Przyjmując następujące oznaczenia: i - ilość iteracji algorytmu, m - ilość mrówek, n - ilość miast; można stwierdzić, że:

- Krok 1 ma złożoność $O(m \cdot n^2)$ - rozmieszczenie m mrówek w losowo wybranych miastach oraz ustawienie wartości feromonu na wszystkich ścieżkach na r_{max} (jest to cecha charakterystyczna dla algorytmu $\mathcal{MM}$AS). Krok ten ma jednak nikły wpływ na złożoność algorytmu (ze względu na jednokrotne wykonanie).

- Krok 2 wykonywany jest i razy, ma więc bezpośredni wpływ na czas wykonywania się programu. Podobnie jak w przypadku algorytmu genetycznego ilość iteracji można ustalić odgórnie, najlepiej jednak dostosować ją indywidualnie do konkretnego problemu. Jednym ze sposobów jest ustalenie ilości iteracji poprzez zadanie górnego limitu czasu w którym ma być wykonywany program, bądź też kosztu trasy jaki jest dla nas zadowalający, po osiągnięciu którego program ma przestać działać.

- Krok 2.1 ma złożoność $O(m \cdot n^2)$ - m mrówek konstruuje swoją trasę, wykonując n kroków (każda z mrówek przechodzi przez wszystkie wierzchołki). Kolejne n jest wynikiem tego, w jaki sposób mrówka wybiera każdy następny wierzchołek na swojej trasie: z pewnym prawdopodobieństwem zależnym od odległości i wartości feromonu na wszystkich ścieżkach prowadzących do wierzchołków jeszcze przez nią nie odwiedzonych.
- Krok 2.2 ma złożoność $O(n^2)$ - sprowadza się tylko do zmniejszenia wartości feromonu na wszystkich krawędziach o pewien stały współczynnik, oraz do zapewnienia, aby wartości feromonu znajdowały się w przedziale $[r_{min}, r_{max}]$.
- Krok 2.3 ma złożoność $O(n)$. W przeciwieństwie do innych algorytmów ACO w algorytmie $\mathcal{MM}$AS tylko jedna (najlepsza) mrówka uaktualnia wartości feromonów, dodając do wszystkich krawędzi należących do skonstruowanej przez nią trasy wartość odwrotnie proporcjonalną do kosztu tej trasy. Poza tym musi zadbać, aby wartości feromonów mieściły się w przedziale $[r_{min}, r_{max}]$.

Podsumowując: algorytm $\mathcal{MM}$AS ma złożoność $O(i \cdot m \cdot n^2)$. Podobnie jak w przypadku algorytmu genetycznego, tylko wartość n jest zadana odgórnie - wartości i oraz m są wartościami umownymi. Wyniki eksperymentalne pokażą, czy lepiej jest wprowadzić więcej mrówek i wykonać więcej iteracji algorytmu, czy też przy mniejszej ilości mrówek wykonać więcej iteracji.

3.3 Wyniki eksperymentalne

Algorytm $\mathcal{MM}$AS został poddany testom dla tych samych danych z TSPLIB, co algorytm genetyczny.

3.3.1 Ilość mrówek

W tablicy 2 zestawiono przykładowe wyniki działania programu dla różnych ilości mrówek (od 20% do 200% ilości wierzchołków), przy jednakowym limicie czasowym (30 s). Wyznaczone w przypadku grafu eil51 trasy są lepsze niż podane w TSPLIB. Widać także wyraźnie w przypadku większych grafów, że bardziej opłacalne jest umieszczanie małej ilości mrówek, zaś zwiększanie liczby iteracji.

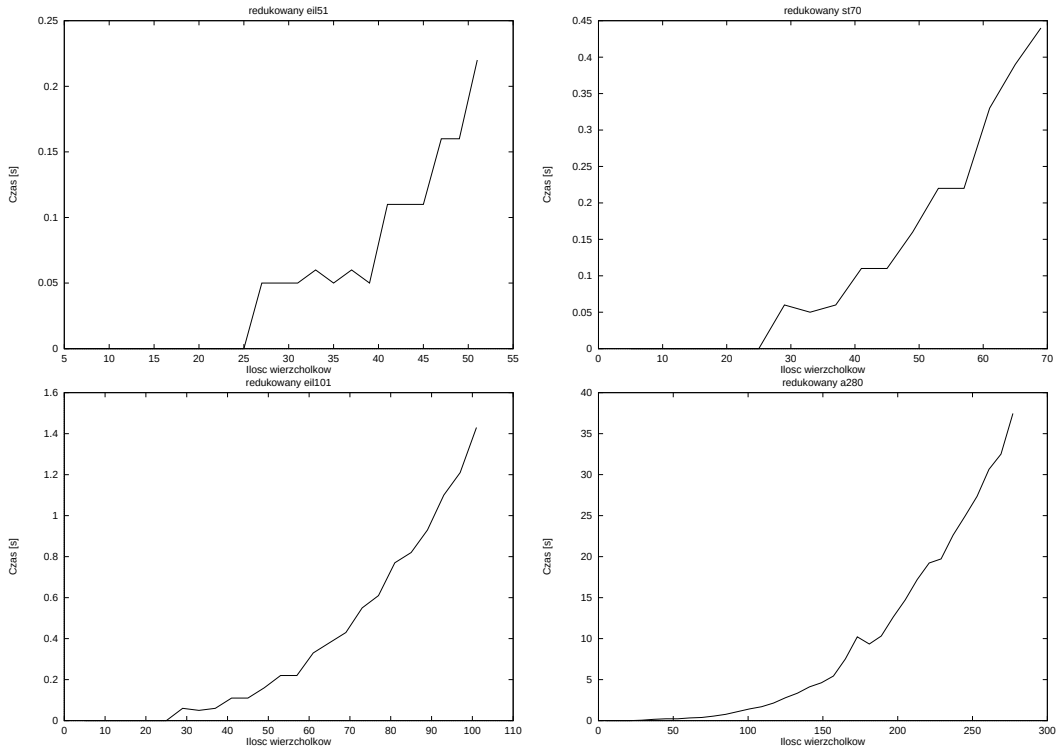
	eil51 (426)		st70 (675)		eil101 (629)	
ρ	iteracje	wynik	iteracje	wynik	iteracje	wynik
20	3001	428	1718	**672**	499	**684**
30	2741	424	1170	696	355	716
40	2223	433	892	683	281	736
50	1868	423	720	681	214	739
60	1549	418	611	704	190	762
70	1381	426	537	691	153	773
80	1170	425	463	688	150	787
90	1092	419	397	697	126	843
100	967	424	361	682	115	846
110	847	428	337	709	95	946
120	775	421	290	686	97	908
130	738	427	291	707	86	1045
140	688	**410**	269	705	77	1094
150	648	423	249	724	71	1098
160	615	426	224	713	64	1099
170	579	424	216	717	66	1044
180	548	427	208	705	63	1149
190	520	430	192	742	61	1175
200	490	428	176	744	55	1105

Tablica 2: Wpływ ilości mrówek na na rozwiązanie. Liczby w nawiasach oznaczają optymalny wynik danego problemu. Liczby pogrubione oznaczają najlepszy uzyskany rezultat.

3.3.2 Złożoność

Testowanie złożoności wykonane zostało podobnie jak w przypadku algorytmu genetycznego. Rysunek 9 przedstawia zależność czasu wykonywania się programu od ilości wierzchołków w grafie. Widać, że zależność ta jest bardziej kwadratowa niż w przypadku algorytmu genetycznego.

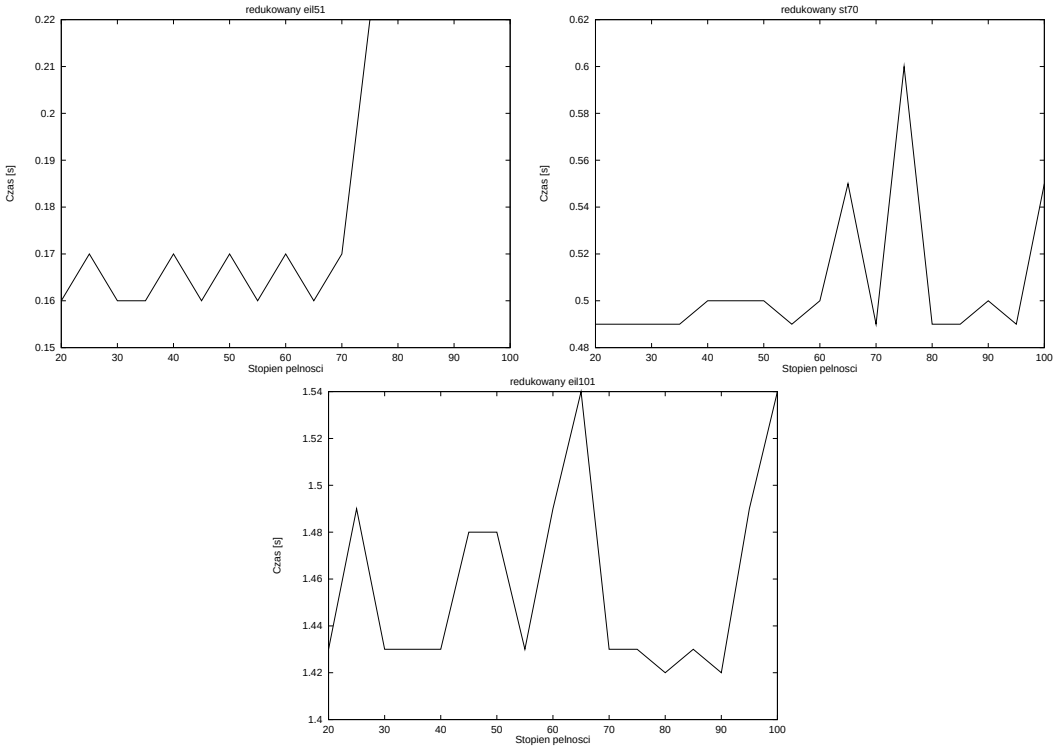
Rysunek 10 przedstawia zależności czasu wykonywania się programu od ilości krawędzi w grafie. Grafy zostały spreparowane podobnie jak w przypadku testowania algorytmu genetycznego - przez wycinanie krawędzi z oryginalnych grafów z TSPLIB. Nie widać jednak żadnej sensownej zależności.



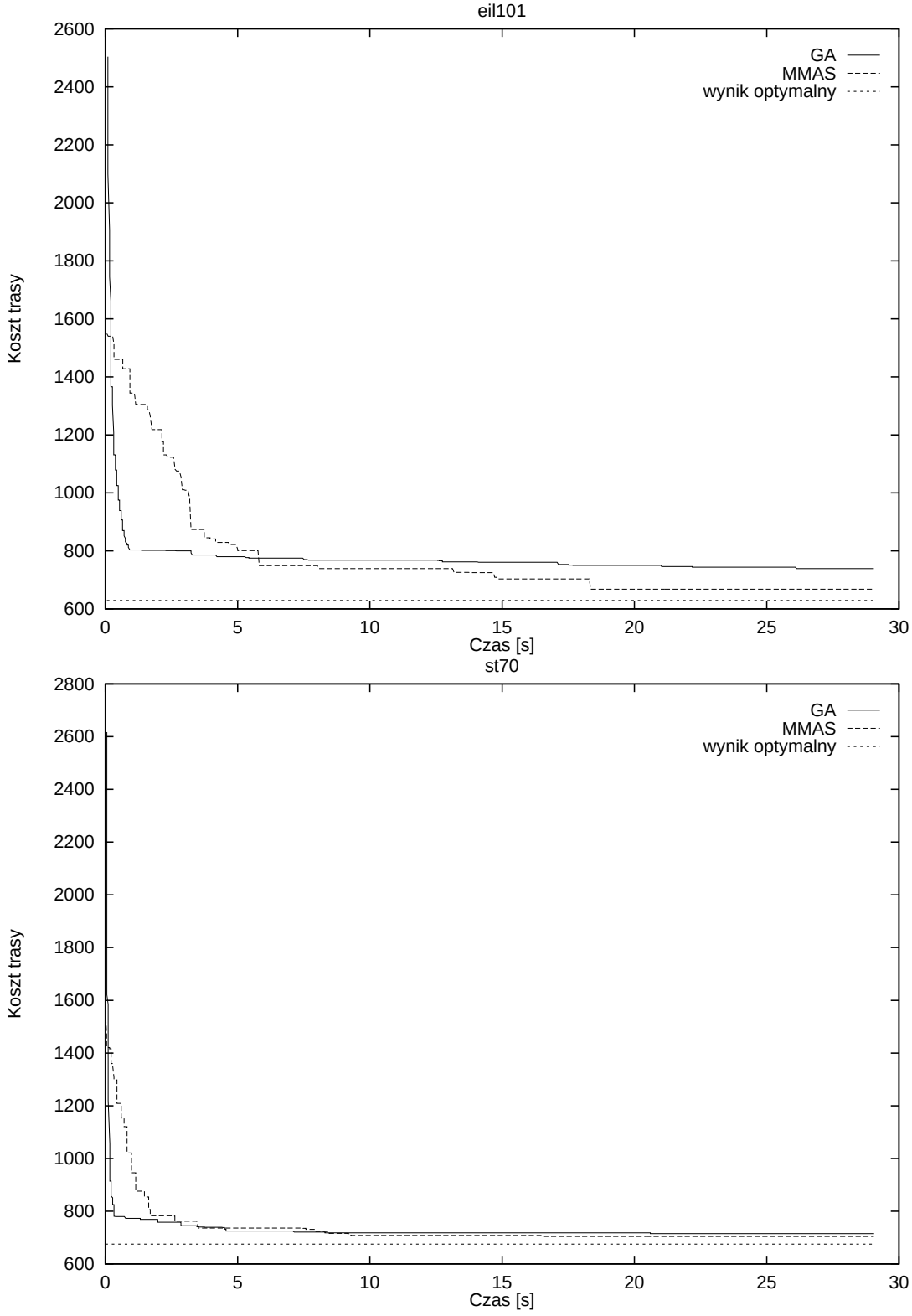
Rysunek 9: Zależność czasu wykonywania się programu od ilości wierzchołków (przy domyślnym ustawieniu parametrów).

4 Porównanie algorytmów

Rysunek 11 przedstawia zachowanie się obu algorytmów w czasie. Warto zauważyć, że algorytm genetyczny rozpoczyna od bardzo złego rozwiązania (wylosowane dowolne rozwiązanie - w większości przypadków zupełnie bezsensowne), podczas gdy algorytm $\mathcal{MM}$AS rozpoczyna dużo niżej (już na samym początku mrówki konstruują trasy zgodnie z odległościami między wierzchołkami). Początkowa zbieżność algorytmu genetycznego jest bardzo szybka, lecz niestety równie szybko wyhamowuje, po osiągnięciu pewnego poziomu zbieżność jest już bardzo wolna. Natomiast algorytm $\mathcal{MM}$AS mimo iż zbiega wolniej, to już wkrótce wyprzedza algorytm genetyczny.



Rysunek 10: Zależność czasu wykonywania się programu od pełności grafu (przy domyślnym ustawieniu parametrów).



Rysunek 11: Porównanie zachowania się algorytmów w czasie.

5 Implementacja

Algorytmy zostały zaimplementowane w języku C++. Poniżej znajdują się krótkie opisy zaimplementowanych klas, ich przeznaczenia oraz podstawowych funkcji bezpośrednio związanych z algorytmami. Ponadto znajduje się tu opis opcji, z jakimi może być uruchomiony każdy z programów

5.1 Algorytm genetyczny

5.1.1 Klasy i funkcje

Klasa **Graph** przechowuje dane grafu, tj. kwadratową macierz kosztów o wymiarach $n \times n$, zawartą w tablicy `weightMatrix`.

Klasa **Tour** reprezentuje jedno rozwiązanie problemu komiwojażera, które przechowywane jest w n-elementowej tablicy `sites`. Ponadto znajduje się tam tablica `indexes`, w której na pozycji i znajduje się indeks pod którym wierzchołek i znajdujes się w tablicy `sites` (pozwala to na szybkie znalezienie następnego wierzchołka w trasie znając tylko numer wierzchołka). Z ważniejszych funkcji należy wymienić funkcję `Crossover()` dokonującą operacji krzyżowania oraz `Mutate()` dokonującą mutacji danego rozwiązania.

Klasa **Population** reprezentuje populację chromosomów. Cała populacja umieszczona jest w tablicy `data`. Ponadto chromosomy umieszczone w tej tablicy połączone są logicznie w łańcuch odsyłaczowy, gdzie wskaźnik do pierwszego elementu łańcucha to `first`. Przeznaczenie tego łańcucha jest następujące: jeśli populacja ma N osobników (`populationSize`) tablica `data` ma $2N$ elementów. Przy inicjalizacji łańcuch przechodzi po kolei przez wszystkie elementy tablicy. Następnie jest sortowany (sam łańcuch odsyłaczowy) i w następnej iteracji pozostaje tylko N pierwszych elementów łańcucha, pozostałe zaś N jest wypełniane potomkami chromosomów znajdujących się w pierwszej części łańcucha. Znów następuje sortowanie itd.

Ważniejsze funkcje klasy **Population** to `Crossovers` tworząca N potomków z N osobników (rodzice wybierani metodą ruletki), `Mutations` dokonująca mutacji pewnej części osobników znajdujących się w tablicy `data`, `Sort` sortująca populację oraz `JudgmentDay` dokonująca reinicjalizacji pewnej części osobników (zwykle większości - pozostaje tylko nieliczna grupa najlepszych).

Ponadto wykorzystywane są pomocnicze klasy **EdgeTriplet** (przy wczytywaniu danych grafu) oraz **TourList** (przy sortowaniu chromosomów w populacji).

Rozwiązanie problemu komiwojażera dokonywane jest w globalnej funkcji `GASolve`, do której przekazywane są odpowiednie opcje podane przez użytkownika (bądź wartości domyślne).

5.1.2 Opcje programu

Opcje decydujące o czasie wykonywania się programu:

`-i` n	ilość iteracji (domyślnie 300)
`-t` n	czas w sekundach, przez który ma działać program (domyślnie doba)
`-b` n	koszt trasy, który spowoduje przerwanie działania programu (domyślnie 0)

Program zatrzymuje się, gdy jeden z tych warunków jest spełniony (domyślne ustawienie limitu czasu na 24 godziny nie przeszkadza temu, że program zatrzyma się już po wykonaniu 300 iteracji algorytmu).

Opcje specyficzne dla algorytmu genetycznego, określające jego zachowanie:

`-r` n	procentowa wielkość „elity” (najlepszych rozwiązań o takim samym koszcie), powodująca reinicjalizacje (domyślnie 10)
`-j` n	procentowa ilość osobnikow przeżywających reinicjalizację (domyślnie 10)
`-p` n	procentowa wielkość populacji w stosunku do ilosci wierzcholkow (domyślnie 200)
`-c` n	n=0 - selekcja losowa n=1 - ruletka (wartość domyślna)

Opcje związane z wyprowadzaniem danych wyjściowych:

`-v` *n*	*n*=0 - (wartość domyślna) na wyjściu pojawiają się dane zgodnie ze specyfikacją projektu, tj. długość całej trasy, a następnie numery kolejnych wierzchołków *n*=1 - wypisywane są numery kolejnych iteracji i koszty trasy przy każdej z nich *n*=2 - wypisywany jest czas (od początku działania programu) oraz koszty trasy
`-s` *n*	numer iteracji, od której wypisywane są dane (jeśli program został uruchomiony z opcją `-v 1` lub `-v 2`) (domyślnie 30)

Opcje te są wykorzystywane przede wszystkim przy tworzeniu wykresów i badaniu zachowania się algorytmu.

5.2 Algorytm $\mathcal{MM}$AS

5.2.1 Klasy i funkcje

Klasa **Graph** przechowuje dane grafu, tj. kwadratowe macierze $n \times n$: `weightMatrix` przechowującą koszty krawędzi grafu oraz `pheromoneMatrix` przechowującą wartości feromonów na każdej z krawędzi. Parowanie feromonów wykonywane jest w funkcji `PheromoneEvaporation`.

Klasa **Ant** to pojedyncza mrówka przechowywująca jedno rozwiązanie problemu komiwojażera. Trasa komiwojażera zapisana jest w tablicy `tour`. Ponadto wykorzystywana jest tablica `visitedVertexes` określająca, które wierzchołki zostały już odwiedzone. Trasa tworzona jest przez funkcję `MakeTour`, która sukcesywnie znajduje kolejne odwiedzane wierzchołki. Uaktualnienie feromonów dokonywane jest w funkcji `UpdatePheromones`.

Ponadto wykorzystywana jest pomocnicza klasa **EdgeTriplet** (przy wczytywaniu danych grafu).

Rozwiązanie problemu komiwojażera dokonywane jest w globalnej funkcji `ACOSolve` do której przekazywane są odpowiednie opcje podane przez użytkownika (bądź wartości domyślne).

5.2.2 Opcje programu

Opcje decydujące o czasie wykonywania się programu:

`-i` *n*	ilość iteracji (domyślnie 50)
`-t` *n*	czas w sekundach, przez który ma działać program (domyślnie doba)
`-b` *n*	koszt trasy, który spowoduje przerwanie działania programu (domyślnie 0)

Działanie tych opcji jest analogiczne do opcji programu realizującego algorytm genetyczny, tj. program zatrzymuje się, gdy jeden z tych warunków jest spełniony (domyślne ustawienie limitu czasu na 24 godziny nie przeszkadza temu, że program zatrzyma się już po wykonaniu 300 iteracji algorytmu).

Opcje specyficzne dla algorytmu $\mathcal{MM}$AS :

`-c` *n*	procentowa ilość mrowek (w stosunku do ilości wierzchołków) (domyślnie 10)
`-e` *n*	procentowy współczynnik parowania feromonu (100 = brak parowania, . . . , 0 = natychmiastowe parowanie) (domyślnie 97)

Opcje związane z wyprowadzaniem danych wyjściowych:

`-v` *n*	*n*=0 - (wartość domyślna) na wyjściu pojawiają się dane zgodnie ze specyfikacją projektu, tj. długość całej trasy, a następnie numery kolejnych wierzchołków *n*=1 - wypisywane są numery kolejnych iteracji i koszty trasy przy każdej z nich *n*=2 - wypisywany jest czas (od początku działania programu) oraz koszty trasy
`-s` *n*	numer iteracji, od której wypisywane są dane (jeśli program został uruchomiony z opcją `-v 1` lub `-v 2`) (domyślnie 30)

Opcje te, podobnie jak analogiczne opcje programu ralizującego algorytm genetyczny są wykorzystywane przede wszystkim przy tworzeniu wykresów i badaniu zachowania się algorytmu.

6 Sprzęt

Wszystkie testy były przeprowadzane na komputerze o następujących parametrach:

- procesor Intel Celeron 433Mhz
- 128MB pamięci operacyjnej
- 128kB pamięci podręcznej
- system operacyjny Windows 98 SE

Bibliografia

[1] Sushil J. Louis, Gong Li; *Genetic Algorithms with Memory for Traveling Salesman Problems*, 1997.

[2] V. M. Kureichick, A. N. Melikhov, V. V. Miagkikh, O. V. Savelev, A. P. Topchy; *Some New Features in Genetic Solution of the Traveling Salesman Problem*, 1996.

[3] Thomas Stützle, Marco Dorigo; *ACO Algorithms for the Traveling Salesman Problem*, 1999.

[4] Thomas Stützle, Holger H. Hoos; *MAX-MINAnt System*, 1999.

[5] Marco Dorigo, Vittorio Maniezzo, Alberto Colorni; *Positive Feedback as a Search Strategy*, 1991.

[6] Eric Bonabeau, Guy Théraulaz; *Mądrość roju*, Świat Nauki 6/2000.

[7] Gerhard Reinelt; *TSPLIB95*