Jan Potocki
nr indeksu 226104

SPRAWOZDANIE

Projektowanie efektywnych algorytmów
Projekt nr 2:
Implementacja i analiza efektywności algorytmu Tabu Search dla wybranego problemu optymalizacji
pon 11:15
18.12.2017

1. Informacje teoretyczne

Problem klasy NP jest problemem, dla którego rozwiązanie można zweryfikować w czasie wielomianowym – natomiast samo znalezienie rozwiązania problemu prawie na pewno wymaga czasu ponadwielomianowego – nie znaleziono dotąd algorytmu o wielomianowej złożoności czasowej. Dla rozwiązania problemów klasy NP można zastosować algorytmy dokładne znajdujące rozwiązanie optymalne, oparte na przykład na metodzie podziału i ograniczeń lub programowania dynamicznego (czego dotyczył poprzedni projekt), jednak w rozsądnym czasie pozwalają one rozwiązywać jedynie niewielkie instancje problemów. Z kolei algorytmy heurystyczne, których przykładem może być przeszukiwanie z zakazami (Tabu Search), pozwalają rozwiązywać w zasadzie dowolnie duże instancje problemów, jednak bez jakiejkolwiek gwarancji optymalności otrzymanego wyniku.

Do implementacji algorytmu Tabu Search w ramach projektu wybrałem problemu komiwojażera.

Problem komiwojażera:
Parametrami zadania są: skończony zbiór miast $C = \{c^1, c^2, \ldots, c^n\}$ oraz odległości d_i z miasta c_i do miasta c_j (dla symetrycznej wersji problemu istnieje wymóg $d_{ij} = d_{ji}$, nie ma go dla wersji asymetrycznej). Należy określić kolejność odwiedzania wszystkich miast ich permutację) $<c_{i[1]}, c_{i[2]}, \ldots, ci_{[n]}>$, aby sumaryczna trasa była jak najkrótsza przy założeniu, że każde miasto zostało odwiedzone dokładnie jeden raz. Wszystkie parametry zadania są liczbami naturalnymi.

2. Opis algorytmu

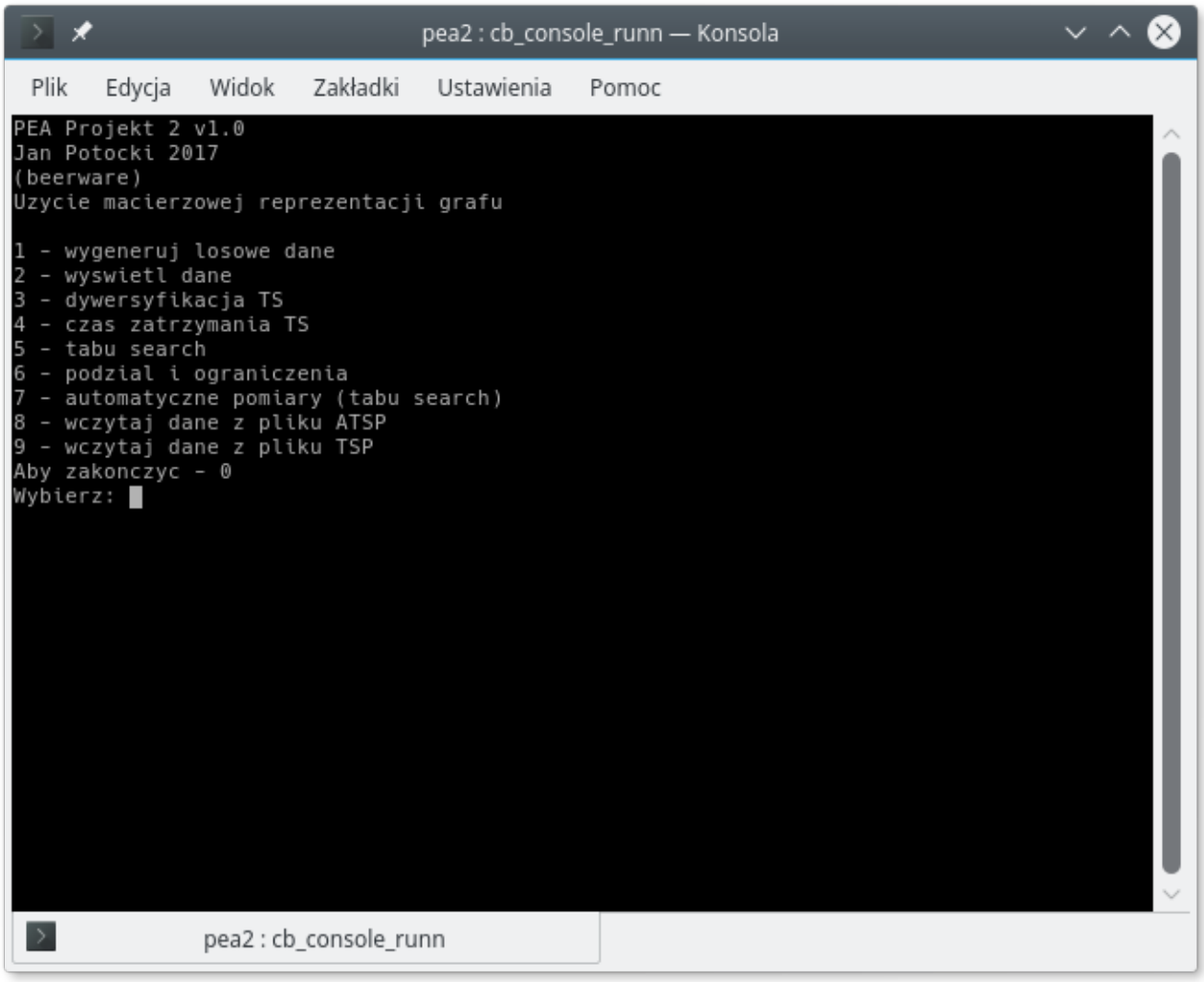
Napisany przeze mnie algorytm w przeszukiwaniu lokalnym wykorzystuje sąsiedztwo typu *swap*. Każdy ruch w tej definicji sąsiedztwa opisany jest parą liczb *(i,j)*, i = 1, …, n, *j = 1, …, n*, przy czym *i* nie może równać się *j* i powoduje zamianę elementu z pozycji *i*-tej z elementem z pozycji *j*-tej. Zamienianymi elementami są wierzchołki grafu w rozwiązaniu (trasie). Na liście tabu przechowywane są pary wierzchołków, zakazując ich ponownej zamiany do czasu upłynięcia kadencji. Kryterium aspiracji pozwala wykonać ruch znajdujący się na liście tabu, jeżeli w jego wyniku powstanie rozwiązanie lepsze od najbardziej optymalnego znalezionego dotychczas. Rozwiązanie początkowe generowane jest za pomocą algorytmu zachłannego. Mechanizm dywersyfikacji generujący nowe rozwiązanie początkowe algorytmem zachłannym z losowym wyborem 3 pierwszych wierzchołków uruchamiany jest po 1000. iteracji bez poprawy rozwiązania. Zatrzymanie algorytmu następuje po określonym czasie, przy czym warunek stopu sprawdzany jest w każdej iteracji po przeglądzie całego sąsiedztwa – co oznacza, że teoretycznie dla dużych instancji problemu pojedynczy przegląd sąsiedztwa może trwać dłużej od zdefiniowanego czasu, a tym samym rzeczywisty czas pracy algorytmu przekroczy ustawienia.

Złożoność obliczeniowa:
Ponieważ algorytm oparty na metodzie Tabu Search jest algorytmem niedeterministycznym, nie można dla niego w całości określić czasowej złożoności obliczeniowej. Można jednak podać złożoność obliczeniową pojedynczego przeglądu całego sąsiedztwa, która dla sąsiedztwa typu *swap* wynosi $O(n^2)$.

3. Opis implementacji algorytmu

Opisany wyżej algorytm zaimplementowałem w języku C++. Graf jest reprezentowany za pomocą własnej struktury danych (stworzonej jeszcze przy pisaniu projektów ze SDiZO): klasy abstrakcyjnej *Graph*, posiadającej dwie implementacje: *ArrayGraph* jako macierz sąsiedztwa i *ListGraph* jako listy sąsiedztwa. Listę tabu zaimplementowałem w rzeczywistości jako tablicę za pomocą kontenera *std::vector z* STL. Do reprezentacji każdej trasy (rozwiązania) wykorzystałem *std::vector*, którego kolejnymi elementami są znajdujące się na niej wierzchołki. Do pomiaru czasu służy własna klasa *Stopwatch*, dokładniej omówiona w kolejnym rozdziale.

4. Środowisko pracy i sposób prowadzenia pomiarów



Ilustracja 1: Menu główne programu

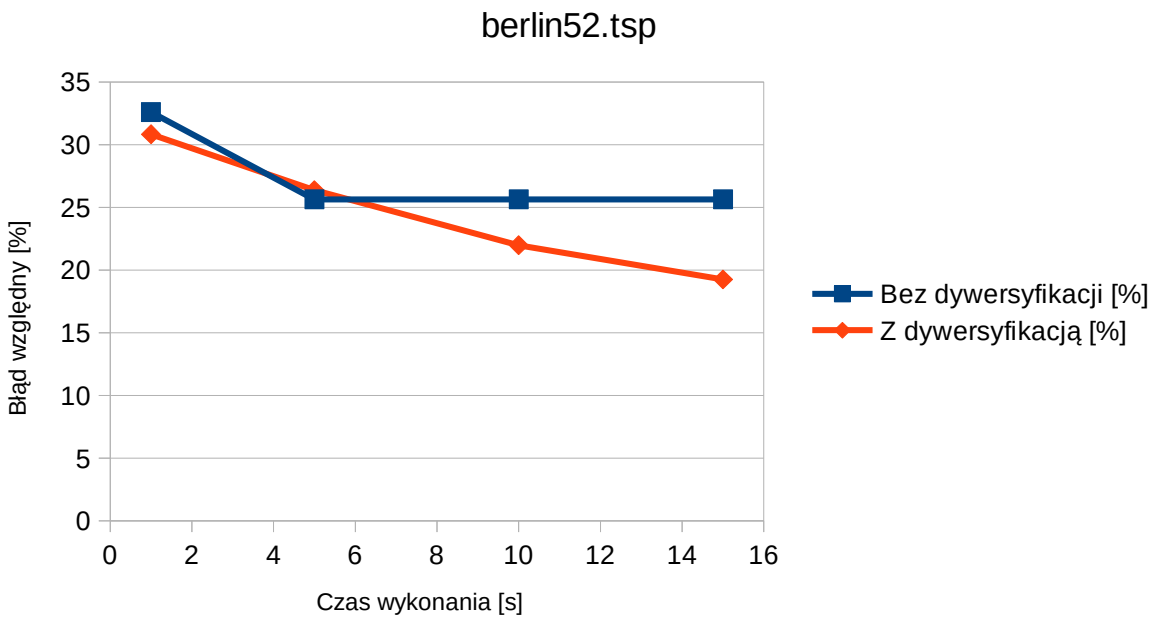
Przy projektowaniu programu wykorzystanego do przeprowadzenia eksperymentu przyjąłem następujące założenia:

- struktury wykorzystywane do reprezentacji danych są alokowane dynamicznie, a struktury pomocnicze (np. zmienne do komunikacji z użytkownikiem) statycznie
- do alokacji i zwalniania pamięci wykorzystuję funkcje *new* i *delete*

- w macierzy i listach sąsiedztwa wierzchołki oraz krawędzie grafu są reprezentowane jako 32-bitowe liczby naturalne (*unsigned*)
- wszystkie pomiary są powtarzane 10 razy, a ich wyniki uśredniane
- pomiary wykonywane są na 3 zestawach danych pochodzących ze strony[1]: *ftv33.atsp*, *berlin52.tsp* i *ftv70.atsp*
- pomiary na każdym z zestawów danych wykonywane są automatycznie z włączoną i wyłączoną dywersyfikacją dla 4 warunków zatrzymania algorytmu: po 1, 5, 10 i 15 sekundach
- kadencja listy tabu w każdym z pomiarów wynosi 40
- program powinien być wieloplatformowy, z tego względu do pomiarów czasu wykorzystuję napisaną przez siebie klasę *Stopwatch*, bazującą na niezależnej od żadnego systemu operacyjnego bibliotece *ctime*

Pomiary przeprowadziłem na komputerze stacjonarnym zbudowanym z wykorzystaniem płyty głównej MSI 870-C45, wyposażonym w czterordzeniowy procesor AMD Phenom II X4 955 BE po overclockingu pracujący z częstotliwością bazową 3.6 GHz i 8 GB pamięci RAM DDR3 pracującej z częstotliwością 1333 MHz w trybie Dual Channel, pod kontrolą 64-bitowego systemu operacyjnego openSUSE Leap 42.3 ze środowiskiem graficznym KDE. Pracowałem w zintegrowanym środowisku programistycznym (IDE) Code::Blocks 16.01, wykorzystując kompilator G++ 4.8 z ustawioną pełną optymalizacją pod kątem szybkości (flaga *-O3*). Przy pomiarach wykorzystałem reprezentację grafu w postaci macierzy sąsiedztwa.

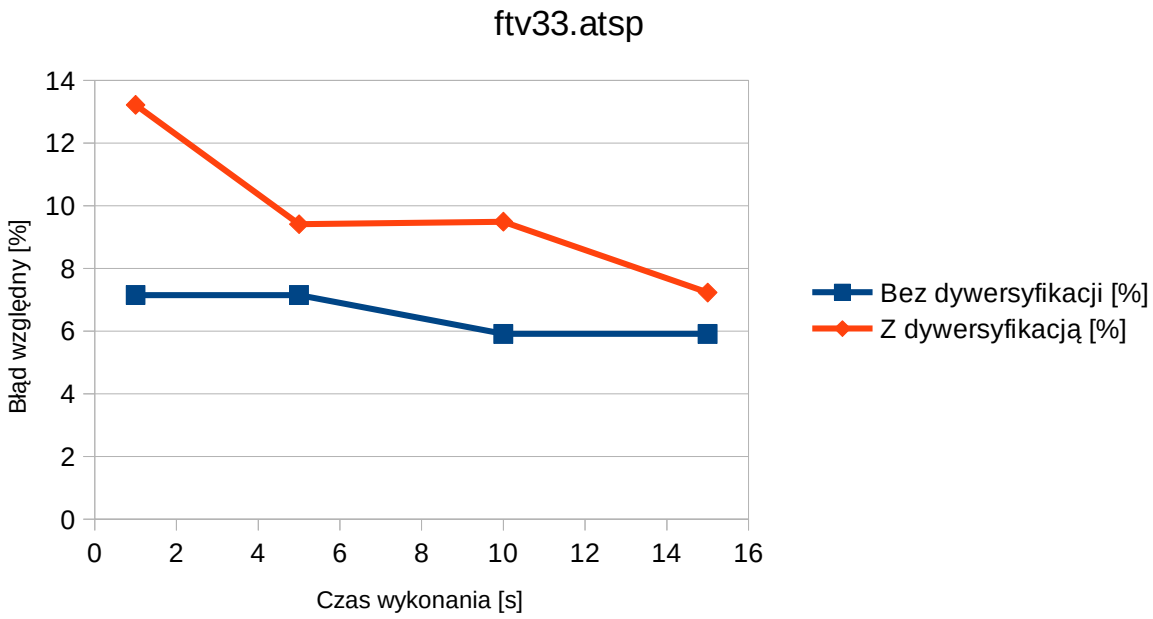
5. Wyniki



berlin52.tsp – 52 miasta, najlepsza znana trasa: 7542

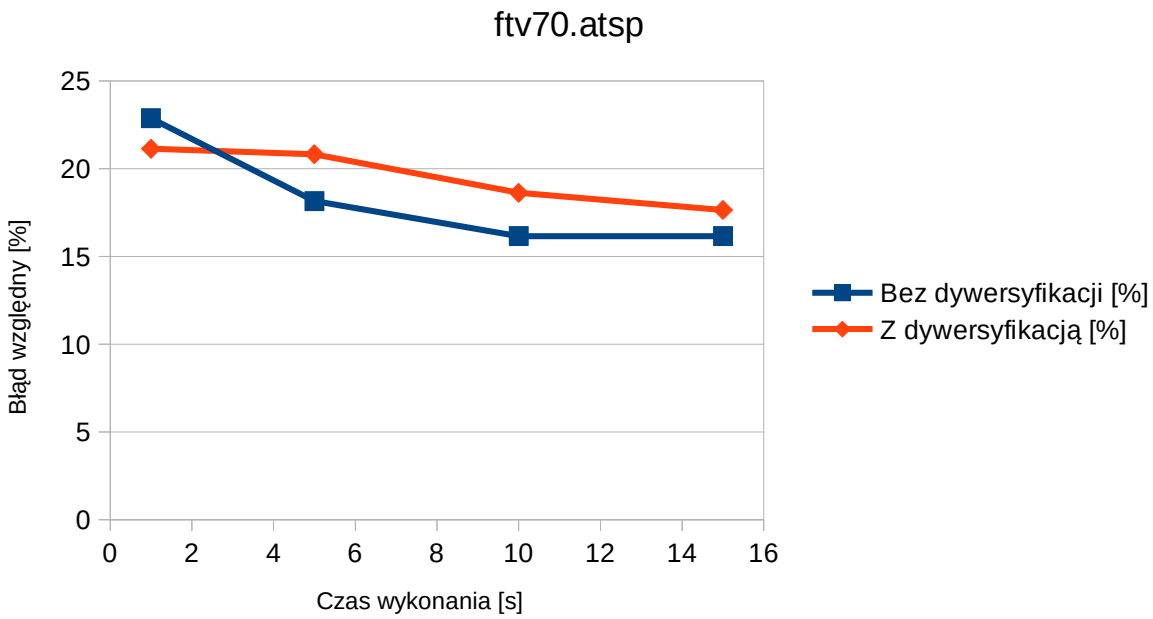
Czas[s]	Bez dywersyfikacji [%]	Z dywersyfikacją [%]	Bez dywersyfikacji	Z dywersyfikacją
1	32,59	30,83	10000	9867
5	25,64	26,37	9476	9531
10	25,64	21,98	9476	9200
15	25,64	19,25	9476	8994

1 http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/



ftv33.atsp – 34 miasta, najlepsza znana trasa: 1286

Czas[s]	Bez dywersyfikacji [%]	Z dywersyfikacją [%]	Bez dywersyfikacji	Z dywersyfikacją
1	7,15	13,22	1378	1456
5	7,15	9,41	1378	1407
10	5,91	9,49	1362	1408
15	5,91	7,23	1362	1379



ftv70.atsp – 71 miast, najlepsza znana trasa: 1950

Czas[s]	Bez dywersyfikacji [%]	Z dywersyfikacją [%]	Bez dywersyfikacji	Z dywersyfikacją
1	22,87	21,13	2396	2362
5	18,15	20,82	2304	2356
10	16,15	18,62	2265	2313
15	16,15	17,64	2265	2294

6. Wnioski

Jedyny zestaw danych, dla których wynik algorytmu z włączoną dywersyfikacją polepszał się wraz ze wzrostem czasu wykonania w stosunku do wyniku bez włączonej dywersyfikacji, jest *berlin52.tsp*. W dwóch pozostałych przypadkach w wyniku dywersyfikacji otrzymałem albo wszystkie rozwiązania gorsze (*ftv33.atsp*), albo wszystkie oprócz pierwszego dla bardzo krótkiego czasu wykonania (*ftv70.atsp*). Jest to spowodowane dobraniem zbyt mocnego – jak się okazało – kryterium dywersyfikacji. Przerywanie poszukiwań i restart algorytmu następuje, jak pisałem wcześniej, po 1000. iteracji bez poprawy – ta liczba w wielu przypadkach jest niewystarczająca, aby wystarczająco dokładnie przeszukać okolicę każdego z kolejnych rozwiązań startowych i z tego powodu lepsze wyniki zapewniało wyłączenie dywersyfikacji. Wtedy przeszukiwana wprawdzie była okolica tylko jednego rozwiązania startowego, ale dokładnie, bo przez cały czas wykonywania algorytmu. Aby się o tym upewnić, testowo dla instancji *ftv33.atsp* uruchomiłem algorytm ze zmienionym kryterium dywersyfikacji na 10000 iteracji bez poprawy, z warunkiem zatrzymania po 300 oraz 500 sekundach i w obu tych przypadkach otrzymałem trasę o długości 1286, czyli najlepszą dotąd znalezioną. Nie miałem już jednak czasu na przeprowadzenie jeszcze raz wszystkich pomiarów ze zmienionym kryterium dywersyfikacji.

Wrocław, 18.12.2017