Projektowanie efektywnych algorytmów Projekt Etap 1

Marcin Kapuściński 248910

Listopad 2021

1 Problem komiwojażera

Problem komiwojażera polega na znalezieniu najtańszej drogi między wszystkimi rozpatrywanymi węzłami grafu - miastami które chce odwiedzić komiwojażer, ostatecznie wracając do miasta początkowego. Oznacza to szukanie minimalnego cyklu Hamiltona - drogi odwiedzającej każdy wierzchołek grafu dokładnie jeden raz, z wyjątkiem początkowego który otwiera i zamyka cykl. Jest to problem NP-trudny, oznacza to że każdy problem klasy NP może być do niego zredukowany w czasie wielomianowym. Na potrzeby projektu, zakłada się że każde miasto połączone jest z każdym - rozpatrywany jest graf pełny. Skutkuje to przestrzenią rozwiązań wielkości $n!$ dla n będącego ilością miast. Graf może być skierowany.

2 Brute force - przegląd zupełny

Przegląd zupełny to metoda polegająca na przeglądzie całej przestrzeni rozwiązań, w poszukiwaniu wyniku optymalnego lub spełniającego postawione wymagania. Często sprowadza się do generowania kolejnych nowych rozwiązań oraz testowania ich względem funkcji celu (*generate and test*). Efekt końcowy uzyskiwany jest po przetestowaniu wszystkich możliwych rozwiązań.

Brute force jest bardzo niewydajną i prostą metodą. Dla instancji problemu wielkości `n` należy wykonać `n` generacji oraz testów rozwiązań, co może być kosztowne. Jeżeli jednak mamy pewność, że `n` będzie stosunkowo małe, implementacja bardziej wydajnych metod może okazać się niepożądana, biorąc pod uwagę ich złożoność i ograniczenia czasowe.

2.1 Implementacja

```
40  Path* BruteForce::findShortestHCycle(int** adjMatrix, int nOfNodesAr, int startNode, int lightestHCycle) {
41      optimalHCycleWeight = lightestHCycle;
42      nOfNodes = nOfNodesAr;
43      bool* passedMatrix = new bool[nOfNodes];
44      for(int node = 0; node < nOfNodes; ++node) {
45          passedMatrix[node] = false;
46      }
47      passedMatrix[startNode] = true;
48      Path currPath(nOfNodes + 1);
49      currPath.addNode(startNode, 0);
50      Path* currLightestPath = new Path(1);
51      currLightestPath->addNode(0, INT_MAX);
52      findShortestHCycleP(adjMatrix, &currPath, passedMatrix, &currLightestPath);
53      delete[] passedMatrix;
54      std::cout << str << std::endl;
55      return currLightestPath;
56  }
```

Rysunek 1: findShortestHCycle(), publiczna

Wszystkie algorytmy zostały zaimplementowane w języku `C++`, z wykorzystaniem biblioteki standardowej. W powyższym listingu miejsce ma przygotowanie danych pomocniczych dla algorytmu. Definiowana i inicjalizowana jest tablica odwiedzonych miast (tablica bool odpowiadających na pytanie ćzy było odwiedzone?", z indeksem odpowiadającym miastu), z zaznaczeniem odwiedzenia miasta początkowego. Tworzona jest ścieżka robocza, miasta będą do niej dodawane i z niej zdejmowane, dodawane jest miasto startowe z kosztem dotarcia do niego równym 0. Tworzony jest startowy najtańszy cykl, o długości jednego miasta o ńieskończonym"koszcie (w implementacji jest to wartość maksymalna dla liczby całkowitej typu `int`). Oczekiwana złożoność algorytmu to $O((n-1)!) = O(n!)$ ponieważ generowane są permutacje $n-1$ wierzchołków.

```
9  void BruteForce::findShortestHCycleP(int** adjMatrix, Path* currPath, bool* traversedMatrix, Path** currLightestPath)
10     if(currPath->currLength == nOfNodes) {
11         currPath->addNode(currPath->nodes[0], adjMatrix);
12         int buffCurrWeight = currPath->weight;
13         if(buffCurrWeight < (*currLightestPath)->weight) {
14             int prd = 100 * (buffCurrWeight - optimalHCycleWeight)/optimalHCycleWeight;
15             if(str.at(0) == 'g'){
16                 str = std::to_string(buffCurrWeight) + ", PRD = " + std::to_string(prd) + "%\n";
17             }
18             std::cout << std::to_string(buffCurrWeight) + ", PRD = " + std::to_string(prd) + "%\n";
19             delete *currLightestPath;
20             *currLightestPath = new Path(*currPath);
21         }
22         currPath->removeTopNode();
23         return;
24     }
25
26     int currBranchingNode = currPath->nodes[currPath->currLength - 1];
27     for(int node = 0; node < nOfNodes; ++node) {
28         if(!traversedMatrix[node] && currBranchingNode != node) {
29             currPath->addNode(node, adjMatrix[currBranchingNode][node]);
30             traversedMatrix[node] = true;
31             findShortestHCycleP(adjMatrix, currPath, traversedMatrix, currLightestPath);
32             currPath->removeTopNode();
33             traversedMatrix[node] = false;
34         }
35     }
36
37     return;
38 }
```

Rysunek 2: findShortestHCycle(), prywatna

Głównym założeniem algorytmu jest rekurencyjna generacja rozwiązań - cykli. Do danych wejściowych należy wybrane miasto początkowe. Oznacza to, że generowane są permutacje $n-1$ pozostałych miast. Po osiągnięciu długości n przez generowaną ścieżkę, dodawane do niej jest miasto startowe, zwieńczając cykl (wyrażenie warunkowe w linii 10., najpewniej usunięte przy optymalizacjach kompilatora). Jeżeli wygenerowany cykl jest tańszy niż do tej pory najtańszy, zastępuje go. Następnie, z generowanej ścieżki zdejmowane jest miasto startowe i następuje powrót do metody wywołującej. Jedyne użyte w implementacji struktury danych to tablice oraz wektory których rozmiar jest rezerwowany - koszt dodawania elementów, nawet niezamortyzowany, jest stały.

Generacja ścieżek zaczyna się w pętli (linia 27.) iterującej po każdym mieście. Jeżeli miasto spełnia wymaganie - nie jest już częścią ścieżki (w kodzie jest zbędne porównanie w warunku). Jeżeli warunek jest spełniony, do generowanej ścieżki dodawane jest rozpatrywane miasto, zostaje oznaczone jako odwiedzone i rekurencyjnie wywoływana jest bieżąca metoda, w celu dalszej generacji ścieżki lub oceny cyklu jeżeli osiągnięta została długość ścieżki n. Takie podejście jest możliwe ponieważ mamy gwarancję grafu pełnego. Do arbitralnie dodawanego miasta początkowego na $n+1$ miejsce cyklu prowadzi droga z każdego miasta. Po rekurencyjnym sprawdzeniu każdego cyklu który można ukończyć z dotychczas zbudowanej ścieżki, wywołanie w linii 31. kończy się. Następuje zdjęcie bieżącego miasta ze ścieżki oraz zaznaczenie go jako nieodwiedzonego. W kolejnej iteracji pętli testowane jest kolejne numerycznie miasto. Po rozpatrzeniu wszystkich n miast na danym poziomie rekurencji (po wygenerowaniu wszystkich możliwych cykli z bieżącej ścieżki), metoda kończy działanie i powraca do generacji na poziomie wcześniejszego miasta. Jeżeli rozpatrywana była ścieżka zawierająca jedynie miasto startowe, algorytm kończy działanie.

3 Branch and bound - podziały i ograniczenia

Metoda podziału i ograniczeń minimalizuje lub maksymalizuje zadaną funkcję celu, przeszukując przestrzeń rozwiązań problemu. Zwiększenie wydajności względem przeglądu zupełnego wynika z:

- **podziałów** przestrzeni rozwiązań na podzbiory. Rekurencyjne dzielenie podzbiorów skutkuje strukturą drzewa jako reprezentacją przestrzeni rozwiązań, z liśćmi będącymi pełnoprawnymi rozwiązaniami i węzłami reprezentującymi podzbiory rozwiązań.
- **ograniczeń** rozpatrywanej przestrzeni rozwiązań przez przypisywanie dolnych lub górnych (minimalizacja lub maksymalizacja) limitów funkcji celu podzbiorom wynikającym z podziałów. Owe limity przypisywane są korzeniom poddrzew w drzewie przestrzeni rozwiązań.

Dla minimalizacji, znając dolną granicę bieżącego węzła drzewa, możemy odrzucić lub rozpatrzyć dane poddrzewo, w zależności od wyniku porównania z granicą górną, stanowiącą najlepsze znalezione rozwiązanie. Aby

szybko zacząć odrzucać duże poddrzewa, warto zainwestować w dobrej jakości startowe górne ograniczenie, w przeciwnym wypadku, na jego ustalenie trzeba czekać aż algorytm osiągnie liść drzewa przestrzeni rozwiązań. Może to znacząco wpłynąć na wydajność, w zależności od strategii przeglądania drzewa. W ramach projektu, branch and bound jest wykonywany do momentu przetestowania wszystkich nieodrzuconych rozwiązań.

3.1 Implementacja

Adaptacja metody branch and bound do problemu komiwojażera skutkuje:

- **podziałami** zrealizowanymi przez tworzenie nowego wierzchołka drzewa przestrzeni rozwiązań, odpowiadającego rozszerzeniu budowanej ścieżki o zadane miasto. Z podziałem wiąże się stworzenie nowej macierzy wag[1], zredukowanej oraz z wykluczonymi krawędziami, odzwierciedlającej dodanie zadanego miasta do ścieżki. Przeprowadzając redukcję macierzy uzyskamy dolne ograniczenie kosztu LB, dla liści poddrzewa (rozwiązań) z korzeniem w rozpatrywanym wierzchołku. Tak stworzony wierzchołek przechowywany jest w kolejce priorytetowej, sortującej po najmniejszej wartości $\frac{LB}{k}$, gdzie k to długość dotychczas zbudowanej ścieżki a wynik to liczba całkowita, zaokrąglona przez obcięcie. Takie kryterium skutkuje strategią przeglądu przestrzeni rozwiązań *best first search*, rozwijającą poddrzewa o najmniejszej średniej długości krawędzi budowanej ścieżki.

- **ograniczeniami** oznaczającymi odrzucenie poddrzewa drzewa przestrzeni rozwiązań, na podstawie porównania LB rozważanego korzenia poddrzewa z ograniczeniem górnym UB. Jeżeli LB korzenia jest większe od UB, mamy pewność, że wszystkie rozwiązania powstałe z tego korzenia będą gorsze niż bieżące najlepsze rozwiązanie.

Oczekiwana złożoność algorytmu w najgorszym przypadku zdaje się być zbliżona do przeglądu zupełnego, o złożoności $O((n-1)!) = O(n!)$, w sytuacji gdy nie mają miejsca ograniczenia.

```
48  int BranchNBound::reduceMatrix(std::vector<std::vector<int>> &matrix, int n) {
49      int reductionCost = 0;
50      //redukcja wierszy
51      int lowestVal;
52      for(int it = 0; it < n; ++it) {
53          lowestVal = findLowestValInRow(matrix, it, n);
54          reductionCost += lowestVal;
55          if(lowestVal != 0) {
56              reduceRow(matrix, it, n, lowestVal);
57          }
58      }
59      //redukcja kolumn
60      for(int it = 0; it < n; ++it) {
61          lowestVal = findLowestValInCol(matrix, it, n);
62          reductionCost += lowestVal;
63          if(lowestVal != 0) {
64              reduceCol(matrix, it, n, lowestVal);
65          }
66      }
67      return reductionCost;
68  }
```

Rysunek 3: ReduceMatrix()

Powyższy wycinek kodu prezentuje redukcję macierzy. Oznacza to znalezienie najmniejszej wartości wiersza, odjęcie jej od każdej wartości komórki tego wiersza, i zapisanie kosztu owej redukcji. Operację tą przeprowadzamy dla każdego wiersza, i potem kolumn. Suma kosztów redukcji wierszy i kolumn to LB dla wierzchołka którego macierz jest redukowana. Redukując wiersz uzyskujemy minimalny koszt wyjścia z odpowiadającego wierzchołka jako koszt redukcji. Redukując kolumny uzyskujemy minimalny koszt wejścia do odpowiadającego wierzchołka. Przeprowadzając redukcję kolumn po redukcji wierszy, uwzględniamy nakładanie się "wyjścia i wejściaña tej samej krawędzi. Całkowity koszt redukcji to minimalny koszt skończonego cyklu.

[1]w kodzie, reprezentacja grafu w postaci macierzy wag nazywana jest macierzą sąsiedztwa *adjMatrix*, poprawniejsze byłoby nazwanie tej zmiennej skrótem zwrotu *weighted adjacency matrix*, "uwagowiona macierz sąsiedztwa". Jednak, w dostępnej mi literaturze anglojęzycznej te nazwy często nie są rozróżniane.

```
6  inline int BranchNBound::findLowestValInRow(std::vector<std::vector<int>> &matrix, int row, int nOfColumns) {
7      int lowestVal = INT_MAX;
8      for(int col = 0; col < nOfColumns; ++col) {
9          if(matrix[row][col] < lowestVal && row != col) {
10             lowestVal = matrix[row][col];
11             if(lowestVal == 0){
12                 break;
13             }
14         }
15     }
16     return (lowestVal == INT_MAX)? 0 : lowestVal;
17 }
```

Rysunek 4: findLowestValInRow()

Funkcja znajdująca najmniejszą wartość w wierszu zadanej macierzy, ignorując przekątną macierzy. Po znalezieniu komórki o wartości 0 funkcja kończy szukanie, zwracając znalezioną wartość. Analogiczna metoda została zaimplementowana dla kolumn.

```
40 inline void BranchNBound::reduceRow(std::vector<std::vector<int>> &matrix, int row, int nOfColumns, int lowestVal) {
41     for(int it = 0; it < nOfColumns; ++it) {
42         if(matrix[row][it] != INT_MAX && it != row) {
43             matrix[row][it] -= lowestVal;
44         }
45     }
46 }
```

Rysunek 5: ReduceRow()

Funkcja redukująca wiersze zadanej macierzy, o zadaną wartość. Redukcji nie podlegają komórki o umownej wartości nieskończoności oraz na przekątnej macierzy. Analogiczna metoda została zaimplementowana dla kolumn.

```
70 void BranchNBound::markEdge(std::vector<std::vector<int>> &matrix, int n, int cityFrom, int cityTo){
71     for(int it = 0; it < n; ++it) {
72         matrix[cityFrom][it] = INT_MAX;//wiersz miasta z ktorego wychodzi
73         matrix[it][cityTo] = INT_MAX;//kolumna miasta do ktorego wchodzi
74     }
75     matrix[cityTo][cityFrom] = INT_MAX;//krawedz powrotna
76 }
```

Rysunek 6: MarkEdge()

Funkcja zaznaczająca odwiedzone miasto w macierzy wag. Wykluczane są krawędzie wychodzące z zadanego miasta, wchodzące do niego, jak i krawędź z poprzedniego miasta do zadanego. Wykluczenie ma miejsce przez ustawienie umownej wartości nieskończoności w komórce.

```
79  std::vector<int> BranchNBound::findShortestHCycle(std::vector<std::vector<int>> &adjMatrix, int nOfNodes, int startNode, int lightestHCycle){
80      std::priority_queue<Node, std::vector<Node>, std::greater<Node>> nodeQ;
81      int UB = INT_MAX;
82      std::vector<int> currBestPath;
83      std::string firstLeafUB = "g";
84
85      nodeQ.push(Node(adjMatrix, nOfNodes, 0));
86      Node buff, branchedNode;
87      while(!nodeQ.empty()){
88          buff = nodeQ.top();
89          nodeQ.pop();
90          if(buff.lowerBound < UB){//dla przedawnionych wierzchołkow w kolejce, o LB >= UB
91              if(buff.lengthOfPath < nOfNodes){
92                  for(int col = 0; col < nOfNodes; ++col) {
93                      if(buff.reducedMatrix[buff.city][col] != INT_MAX && col != startNode) {//branching
94                          branchedNode = Node(&buff, col, adjMatrix);
95                          if(branchedNode.lowerBound < UB) {
96                              nodeQ.push(branchedNode);
97                          }
98                      }
99                  }
100             }else {
101                 branchedNode = Node(&buff, 0, adjMatrix);
102                 if(branchedNode.costOfPath < UB) {
103                     UB = branchedNode.costOfPath;
104                     int prd = 100 * (UB - lightestHCycle)/lightestHCycle;
105                     std::cout << UB << ", PRD = " << prd << "%\n";
106                     if(firstLeafUB.at(0) == 'g') {
107                         firstLeafUB = std::to_string(UB) + ", PRD = " + std::to_string(prd) + "%\n";
108                     }
109                     currBestPath = branchedNode.path;
110                 }
111             }
112         }
113     }
114     std::cout << firstLeafUB << std::endl;
115     return currBestPath;
116 }
```

Rysunek 7: findShortestHCycle()

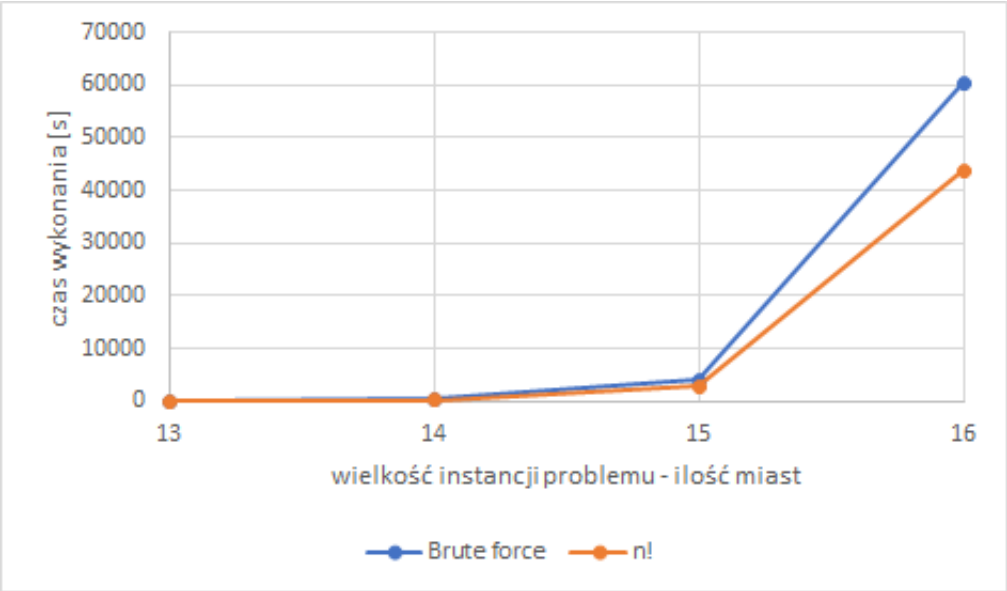
Algorytm wykonuje się dopóki kolejka priorytetowa korzeni poddrzew drzewa przestrzeni rozwiązań nie jest pusta. Na początku do kolejki wkładany jest korzeń reprezentujący miasto startowe. W pętli z kolejki zdejmowany jest najbardziej obiecujący korzeń, jeżeli się przedawnił i znaleziono dyskwalifikujący go UB, nie jest rozpatrywany. Jeżeli jest rozpatrywany i przypisana my ścieżka ma długość mniejszą od n, następuje podział. Jeżeli wierzchołki powstałe w podziale cechuje LB mniejszy od UB, wkładane są do kolejki. Jeżeli wierzchołek cechuje ścieżka długości n, następuje podział tworzący jedynie liść, dodając tylko miasto startowe do ścieżki, kończąc cykl. Jeżeli całkowity koszt cyklu jest mniejszy od aktualnego UB, znaleziono nowe najlepsze rozwiązanie, należy je zapamiętać i zaktualizować UB.

4 Pomiary

Poprawność działania algorytmu zakłada się jeżeli wyprodukowane cykle mają koszt przynajmniej tak niski jak podane w plikach testowych optima. Kierując się tym kryterium, zaimplementowane algorytmy uznaje się za poprawne. Pomiary zostały wykonane po skompilowaniu z opcją optymalizacji 3 poziomu, na procesorze o 8 rdzeniach logicznych oraz maksymalnym taktowaniu 3.6 GHz. Jednocześnie przeprowadzane były 2 pomiary, z minimalnym obciążeniem procesora przez inne programy. Ze względu na intensywność czasową oraz determinizm algorytmów, pomiary dla poszczególnych wielkości instancji problemu zostały wykonane jednokrotnie. Zakłada się miarodajność wyników ze względu na duży czas wykonania, praktycznie niwelujący wpływ sporadycznych wahań w poziomie użycia procesora przez system, na ostateczny wynik. Wyjątkiem są pomiary o czasie wykonania poniżej 100 sekund, te zostały powtórzone 10 razy i zaprezentowane zostały czasy uśrednione.

4.1 Brute force

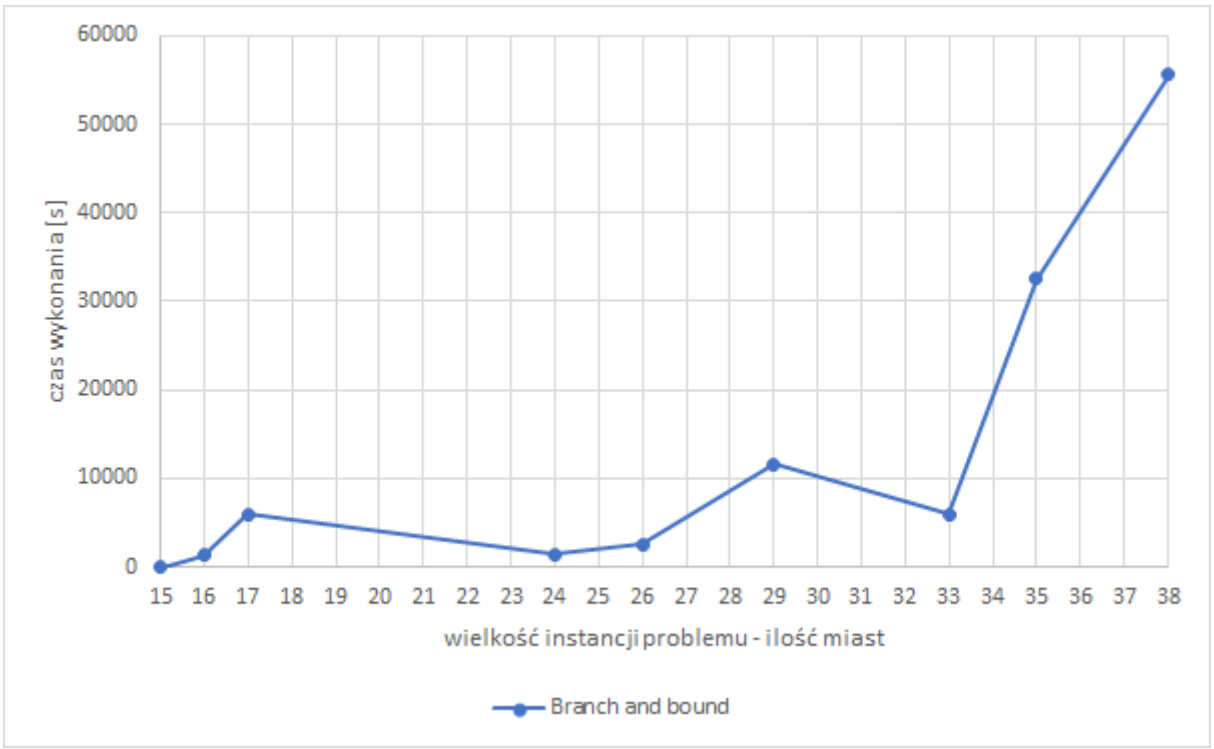
n	13	14	15	16
czas[s]	21	271	3980	60473
$\frac{czas_n}{czas_{n-1}}$		12.70	14.69	15.19



Rysunek 8: czasy wykonania alg. brute force

4.2 Branch and bound

n	15	16	17	24	26	29	33	35	38
czas[s]	1	1328	6000	1437	2602	11630.5	5933	32579	55558
$\frac{czas_n}{czas_{n-1}}$		1328.00	4.52	0.24	1.81	4.47	0.51	5.49	1.71



Rysunek 9: czasy wykonania alg. branch and bound

5 Wnioski

5.1 Brute force

Z powodzeniem można przyrównać zależność czasu wykonania od wielkości instancji problemu, do wzrostu eksponencjalnego, zgadzającego się z przewidywaniami.

5.2 Branch and bound

Udało się uzyskać zależność czasu wykonania od wielkości instancji problemu znacząco lepszą od wykładniczej. Świadczy to o opłacalności zabiegów ograniczania rozpatrywanej przestrzeni rozwiązań, względem prostego przeglądu zupełnego. Duże wzrosty dla n równego 17(m17.atsp) oraz 29(bays29.tsp) wynikają zapewne ze stosunkowo bardzo małych różnic w wartości LB między korzeniami, ze względu na zbliżenie kosztów krawędzi. Skutkuje to zniekształceniem *best first search* do przeszukiwania po szerokości. Widoczne to było w znacznym zużyciu pamięci przez program, w porównaniu do obliczeń dla innych instancji (rzędy wielkości wyższe). W takim wypadku, kryterium sortowania kolejki okazało się mało pomocne, zmiana wyliczania średniej do liczby zmiennoprzecinkowej nie usprawniło wyników. Spełnione zostało oczekiwanie wydajności lepszej niż wykładnicza, jednak ocena zależności czasu wykonania od wielkości instancji problemu jest bardzo trudna. Zdaje się, że największy wpływ na czas wykonania dla przedstawionej implementacji ma charakterystyka instancji problemu, zbliżenie kosztów ścieżek. Dla algorytmu o lepszym sposobie liczenia LB lub priorytetu dla kolejki, ten problem może być w pewnym stopniu zniwelowany.