Politechnika Wrocławska

Wydział Informatyki i Telekomunikacji

Projektowanie efektywnych algorytmów

Projekt nr 2 - analiza efektywności metody podziału i ograniczeń (B&B) dla asymetrycznego problemu komiwojażera (ATSP)

semestr zimowy 2023/2024

Autor:
Eryk Mika 264451

Prowadzący:
Dr inż. Marcin Łopuszyński

Spis treści

1. Wstęp teoretyczny

Zgodnie z informacjami przedstawionymi w sprawozdaniu z projektu nr 1, asymetryczny problem komiwojażera jest problemem bardzo trudnym pod względem obliczeniowym. W tym opracowaniu zostanie omówione rozwiązanie tego problemu z wykorzystaniem metody podziału i ograniczeń (ang. *branch and bound*, B&B).

W metodzie B&B stosujemy ograniczenia w celu wyeliminowania rozwiązań, o których wiemy, że są nieobiecujące. Eliminacja i wybór rozwiązań dokonywany jest za pomocą drzewa przestrzeni stanów, które reprezentuje wszystkie ścieżki, według których algorytm może zostać wykonany. Węzły drzewa reprezentują częściowe (lub całkowite – w przypadku liści drzewa) rozwiązania problemu. Węzeł nieobiecujący definiujemy jako taki, którego granica jest gorsza od aktualnie najlepszego rozwiązania. Granicę natomiast definiujemy jako liczbę, która wyznacza ograniczenie – wartość rozwiązania, jakie może być uzyskane poprzez rozwinięcie danej ścieżki w drzewie. W przypadku problemu ATSP metoda B&B sprowadza się do przeszukiwania drzewa rozwiązań, którego liście zawierają możliwe rozwiązania (kompletne cykle Hamiltona). Możliwe jest zaimplementowanie różnych wariantów algorytmów rozwiązujących problem ATSP metodą B&B. W zdecydowanej części różnią się one strukturami danych użytymi do realizacji algorytmu i sposobem przeszukiwania drzewa rozwiązań. Należą do nich:

- algorytm *breadth-first search* (przeszukiwanie wszerz) - z wykorzystaniem kolejki FIFO (ang. *First In First Out*),
- algorytm *depth-first* (przeszukiwanie w głąb) - z wykorzystaniem stosu – zastosowany w projekcie,
- algorytm *best-first* (najpierw najlepszy) - z wykorzystaniem kolejki priorytetowej, w której priorytetem elementów kolejki – węzłów drzewa - są ich ograniczenia dolne.

Istotną kwestią związaną z omówieniem metody B&B jest odróżnienie ograniczenia dolnego (ang. *lower bound*) oraz ograniczenia górnego (ang. *upper bound*). Ograniczenie dolne rozumiane jest jako najmniejsza możliwa wartość rozwiązania, jaka może zostać uzyskana z rozwinięcia danego węzła, natomiast ograniczenie górne rozumiane jest jako najlepsze znane (w danym momencie wykonywania algorytmu) rozwiązanie określonego problemu. Porównywanie ograniczeń dolnych węzłów drzewa przestrzeni stanów z ograniczeniem górnym może być wykorzystane do wykluczania określonych ścieżek – rozwiązań [1].

Oszacowanie złożoności czasowej algorytmu realizującego metodę B&B jest zadaniem trudnym. W najbardziej pesymistycznym przypadku jego złożoność będzie taka sama, jak algorytmu przeglądu zupełnego (**O(n!)**)[2]. Złożoność średnia ma zazwyczaj niższy rząd wielkości[3], co istotnie zależy od konkretnego sposobu implementacji algorytmu.

[1] https://en.wikipedia.org/wiki/Branch_and_bound
[2] http://vigir.missouri.edu/~gdesouza/Research/Conference_CDs/IEEE_ICCV_2009/contents/pdf/iccv2009_244.pdf
[3] https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/

2. Opis implementacji algorytmu

W celu analizy efektywności algorytmu został napisany, podobnie jak w projekcie nr 1, program w języku *C++* z wykorzystaniem obiektowego paradygmatu programowania. Najważniejszymi częściami programu z punktu widzenia realizacji projektu są klasy *Graph*, *BnBNode* oraz *BnBStack*, które zawierają pola (atrybuty) oraz metody odpowiedzialne za realizację algorytmu. Szczegółowe wyjaśnienie poszczególnych fragmentów kodu zawarte jest również w plikach źródłowych.

2.1. Klasa *Graph*

Klasa Graph jest główną klasą programu, która jest odpowiedzialna za przechowywanie struktury i metod grafu, na którym wykonywany jest algorytm realizujący metodę B&B dla problemu ATSP. Pola prywatne klasy – dwuwymiarowa tablica *std::vector matrix* oraz size są użyte do przechowywania długości krawędzi w postaci macierzy kwadratowej – kosztów - *matrix* stopnia *size*. Oba pola przechowują liczby stałoprzecinkowe typu *int*. W macierzy komórka o współrzędnych *i*, *j* zawiera odległość pomiędzy wierzchołkami *i* i *j*.

Zaimplementowano konstruktory (domyślny, generujący losową instancję problemu o rozmiarze N, wczytujący instancję z pliku tekstowego, przeładowany operator przypisania oraz metodę wypisującą graf (macierz) na ekran. Konstruktor generujący losową instancję problemu wykorzystuje funkcję *rand()* z biblioteki *<random>* do generowania liczb pseudolosowych – długości krawędzi po uprzednim zainicjalizowaniu generatora za pomocą wywołania funkcji *srand()* z określonym ziarnem generatora jako argument.

2.2. Klasa *Graph* - obliczenie ograniczenia górnego (*upper bound*)

Zaimplementowana została metoda *calcUpBnd()*, która jest użyta do obliczenia wstępnego ograniczenia górnego kosztu dla danego problemu ATSP na początku działania algorytmu w sposób **zachłanny**. Algorytm metody może być zapisany w postaci listy kroków:

1. Algorytm rozpoczynamy od ustalenia kosztu jako 0 oraz ustawienia węzła obecnego na węzeł początkowy, który jest przyjęty, podobnie jak korzeń drzewa stanów w metodzie B&B, jako 0. Dodajemy węzeł początkowy do tablicy *visited*.
2. Przeszukując wiersz macierzy problemu ATSP o indeksie obecnego węzła znajdujemy minimum w każdym wierszu – wartość w kolumnie o indeksie węzła jeszcze nieodwiedzonego (który nie jest zapisany w tablicy *visited*).
3. Wyznaczony w poprzednim kroku nieodwiedzony jeszcze węzeł dodajemy do tablicy *visited* oraz ustawiamy go jako obecny węzeł. Dodajemy wartość krawędzi łączącej wcześniejszy oraz obecny węzeł do kosztu.

4. Kroki 2-3 powtarzamy aż rozmiar tablicy *visited* nie będzie równy rozmiarowi problemu – stopniowi macierzy.
5. Na końcu dodajemy do kosztu wartość krawędzi łączącej ostatni dodany do tablicy *visited* wierzchołek z wierzchołkiem początkowym (0). Otrzymujemy szukane ograniczenie górne.

2.3. Klasa *Graph* – metoda *timeBranchAndBoundATSP()*

Jest to główna metoda programu, która służy do rozwiązania problemu ATSP metodą B&B. Algorytm rozpoczyna się od utworzenia w pamięci korzenia drzewa (a więc węzła) – obiektu klasy *BnBNode* oraz inicjalizacji stosu przechowującego wskaźniki do węzłów. Obliczamy wstępne ograniczenie górne zachłannie korzystając z metody *calcUpBnd()*. Następnie w pętli, dopóki stos nie jest pusty, zdejmujemy wskaźnik do węzła ze szczytu stosu i ustawiamy go jako obecny (*current*). Sprawdzamy, czy obecny węzeł jest liściem i wartość jego rozwiązania jest lepsza od ograniczenia górnego, jeżeli tak – zapisujemy nową najlepszą ścieżkę oraz korygujemy ograniczenie górne. Usuwamy węzeł i kontynuujemy pętlę od kolejnego elementu na stosie. W przeciwnym razie, dla każdego możliwego potomka danego węzła sprawdzamy, czy jest on obiecujący – jego ograniczenie dolne nie jest większe od obecnego ograniczenia górnego. Jeżeli tak jest, umieszczamy danego potomka na stosie, a w przeciwnym razie go usuwamy. Na końcu każdego wykonania pętli dealokujemy pamięć po obecnym węźle. Omówiona główna część algorytmu została przedstawiona na Rysunek 2.1.

Rysunek 2.1 Główna część algorytmu realizującego metodę B&B

```
117     // Dopoki stos nie jest pusty
118     while(!st.isEmpty())
119     {
120         // Zdejmujemy i usuwamy szczytowy element stosu
121         BnBNode* current = st.peek();
122         st.pop();
123 
124         /*
125             Jezeli wierzcholek to lisc i jego koszt jest mniejszy od gornej granicy,
126             zapisujemy nowa najlepsza sciezke i gorna granice. Dealokujemy pamiec. Kontynuujemy algorytm od kolejnego
127             wierzcholka na stosie.
128         */
129         if(current->isLeaf() && current->getCost()<=upBound)
130         {
131             path = current->getPath();
132             upBound = current->getCost();
133             delete current;
134             continue;
135         }
136 
137         // Indeks obecnego wierzcholka
138         int currentNode = current->getNode();
139 
140         /*
141             Iterujemy po wszystkich mozliwych potomkach danego wezla. Jezeli jego koszt jest mniejszy niz wartosc
142             gornego ograniczenia (jest "obiecujacy"), umieszczamy go na stosie. W przeciwnym wypadku - usuwamy.
143         */
144         for(int i=0; i<size; i++)
145         {
146             int edgeLen = current->getMatrix()[currentNode][i];
147             if(edgeLen>=0)
148             {
149                 BnBNode* child = new BnBNode(i, current->getMatrix(), currentNode, current->getCost() + edgeLen, current->getPath());
150                 if(child->getCost()<=upBound)
151                     st.push(child);
152                 else delete child;
153             }
154         }
155         // Zwalniamy pamiec po obecnym wierzcholku
156         delete current;
157     }
```

Źródło: opracowanie własne

Na końcu metody wyświetlana jest najlepsza znaleziona ścieżka, jej koszt oraz czas wykonywania algorytmu, który jest także zwracany z metody.

2.4. Klasa *Graph* – metoda *timeBranchAndBoundATSPlimited()*

Jest to wariant metody z punktu 2.3, ale z ograniczeniem czasowym wykonywania algorytmu. Co 1000 wykonań pętli *while* obecny czas wykonywania algorytmu jest porównywany z limitem określonym w pliku nagłówkowym *Graph.h* (w milisekundach). Jeżeli czas jest przekroczony (co sygnalizowane jest poprzez zmianę flagi *timeout*), algorytm jest przerywany i z metody zwracana jest liczba ujemna. Sprawdzanie warunku przerwania algorytmu co określoną liczbę iteracji ma na celu zapobieżenie nadmiernemu wpływowi pomiaru czasu na czas wykonywania algorytmu – całość odbywa się w sposób jednowątkowy.

2.5. Klasa *BnBNode*

Obiekt klasy *BnBNode* reprezentuje pojedynczy węzeł drzewa stanów. W polu prywatnym *std::vector matrix* przechowuje, podobnie jak obiekt klasy *Graph*, macierz kosztów, ale zmodyfikowaną – zredukowaną dla przypadku konkretnego węzła drzewa. Pole prywatne std::vector *path* przechowuje ścieżkę, będącą częściowym rozwiązaniem węzła. Pola prywatne *cost* oraz *node* typu int przechowują odpowiednio koszt związany z danym węzłem oraz indeks węzła. Wskaźnik *BnBNode* next* wskazuje na następny element stosu *BnBStack*.

Konstruktor klasy przyjmuje argumenty, które odpowiednio oznaczają indeks węzła będącego ostatnim odwiedzonym węzłem dla danego częściowego rozwiązania, macierz kosztów rodzica danego węzła, indeks wiersza, który ma być wykluczony, koszt poprzedniego węzła oraz jego ścieżka. W konstruktorze następuje wykluczenie (ustawienie jako -1) wiersza o indeksie potomka danego węzła oraz kolumny o indeksie obecnego węzła. Zabroniony jest także powrót do korzenia poprzez ustawienie długości krawędzi łączącej obecny węzeł z korzeniem jako -1. W konstruktorze następuje także przypisanie kosztu obecnego węzła jako suma kosztu poprzedniego węzła oraz wartości zwróconej z metody redukującej macierz – *reduceMatrix()*. Zostały zaimplementowane także metody – gettery – zwracające wartości pól prywatnych klasy lub referencje do nich oraz metoda *isLeaf()*, która służy do sprawdzenia, czy węzeł jest liściem, czyli zawiera kompletne rozwiązanie problemu ATSP. Setter *setNext()* użyty jest do ustawiania wskaźnika (*next*) na następny węzeł na stosie.

2.6. Klasa *BnBNode* – metoda *reduceMatrix()*

Metoda ta jest wykorzystana do obliczenia ograniczenia dolnego poprzez algorytm redukcji macierzy[4] - macierz redukowana jest w taki sposób, aby w każdym wierszu i każdej kolumnie znalazło się przynajmniej jedno zero, a całkowity koszt redukcji jest zwracany z metody.

[4] https://cs.pwr.edu.pl/zielinski/lectures/om/mow10.pdf

Przechodząc po wszystkich wierszach i kolumnach, znajdujemy dla każdego wiersza i kolumny **nieujemne** minimum, które następnie odejmujemy od danego wiersza/kolumny. Suma tych minimalnych wartości jest całkowitym kosztem redukcji. Nieuwzględnianie ujemnych wartości ma istotne znaczenie ze względu na **pomijanie wykluczonych ścieżek**. Implementację metody przedstawia Rysunek 2.2.

Rysunek 2.2 Metoda redukująca macierz – zwracająca koszt redukcji.

```
39  int BnBNode::reduceMatrix()
40  {
41      int sumOfReduction = 0;
42
43      // Petla do przechodzenia jednoczesnie po wierszach i kolumnach macierzy wierzcholka
44      for(int i=0; i<size; i++)
45      {
46          int rowMin = INT_MAX;
47          int colMin = INT_MAX;
48          // Znajdujemy minimum w wierszu i kolumnie, pomijamy ujemne wartosci (np. z przekatnej).
49          for(int j=0; j<size; j++)
50          {
51              if(matrix[i][j] >= 0 && matrix[i][j] < rowMin) rowMin = matrix[i][j];
52              if(matrix[j][i] >= 0 && matrix[j][i] < colMin) colMin = matrix[j][i];
53          }
54          // Mozliwe, ze 0 - wczesniej pozostalo MAX
55          rowMin = ((rowMin!=INT_MAX) ? rowMin : 0);
56          colMin = ((colMin!=INT_MAX) ? colMin : 0);
57          // Koszt - suma redukcji wierszy row i kolumn col
58          sumOfReduction += rowMin + colMin;
59
60          // Redukujemy macierz
61          for(int j=0; j<size; j++)
62          {
63              matrix[i][j] -= rowMin;
64              matrix[j][i] -= colMin;
65          }
66      }
67      return sumOfReduction;
68  }
```

Źródło: opracowanie własne

2.7. Klasa *BnBStack*

Klasa ta jest implementacją stosu (kolejki typu *LIFO*) służącej do przechowywania wskaźników do obiektów *BnBNode*. Klasa zawiera jedno pole prywatne *top*, które przechowuje wskaźnik na szczyt stosu. Zostały napisane metody realizujące podstawowe funkcje stosu:

- peek() – służąca do odczytywania wartości na szczycie stosu,
- push() – służąca do umieszczania elementu na stosie,
- pop() – służąca do zdejmowania elementu ze stosu,
- isEmpty() – służąca do sprawdzenia, czy stos jest pusty.

Implementacja została przedstawiona na Rysunek 2.3. Modyfikacja stosu polega na odpowiedniej zmianie wskaźnika *top*.

Rysunek 2.3 Implementacja metod stosu BnBStack

```
3   // Inicjalizujemy stos - szczyt jako NULL
4   BnBStack::BnBStack()
5   {
6       top = NULL;
7   }
8
9   // Dodajemy wskaznik do stosu, zmieniamy szczyt
10  void BnBStack::push(BnBNode* node)
11  {
12      node->setNext(top);
13      top = node;
14  }
15
16  // Zwracamy wskaznik do elementu na szczycie stosu
17  BnBNode* BnBStack::peek() const
18  {
19      if(top==NULL) return NULL;
20      return top;
21  }
22
23  // Czy stos jest pusty
24  bool BnBStack::isEmpty() const
25  {
26      return top==NULL;
27  }
28
29  // Zamieniamy szczyt stosu na element pod szczytowym elementem
30  void BnBStack::pop()
31  {
32      if(top==NULL) return;
33      top = top->getNext();
34  }
35
```

Źródło: opracowanie własne

3. Przykład praktyczny działania algorytmu „krok po kroku”

W celu przedstawienia działania algorytmu został wybrany następujący problem ATSP o rozmiarze $N = 4$:

Rysunek 3.1

	0	**1**	**2**	**3**
0	−1	2	7	3
1	1	−1	6	4
2	5	2	−1	8
3	6	3	1	−1

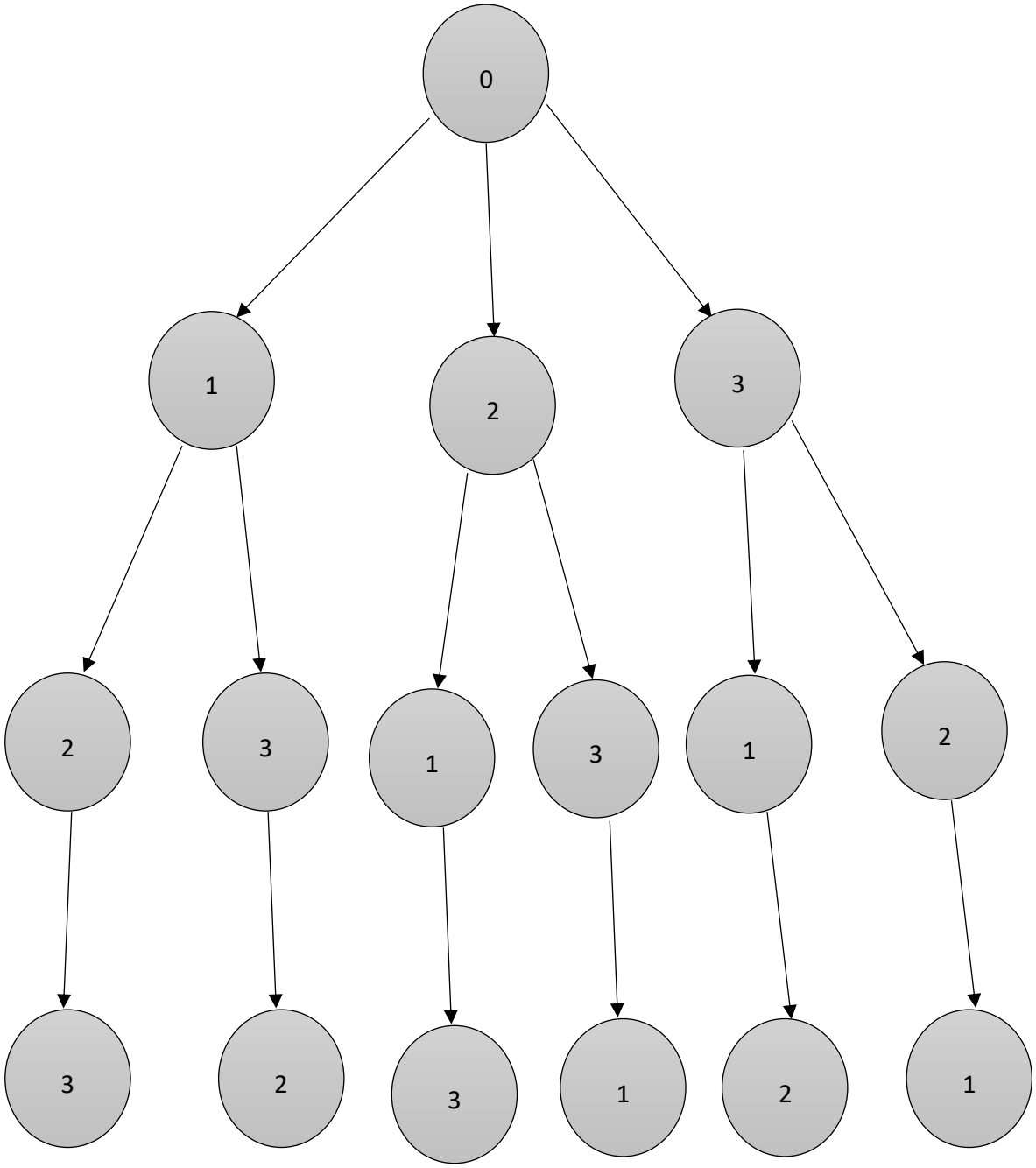
Źródło: opracowanie własne

Jest to reprezentacja problemu w postaci macierzy sąsiedztwa grafu – tak samo jak w przypadku kodu programu. Identyczna jest numeracja (pogrubione cyfry) wierszy i kolumn – od 0, która odpowiada numeracji wierzchołków grafu. Element macierzy w i-tym wierszu i j-tej kolumnie to wartość długości krawędzi łączącej wierzchołki *i* i *j*.

3.1. Reprezentacja drzewa przestrzeni stanów dla problemu

Wszystkie możliwe ścieżki w grafie, a tym samym wszystkie możliwe rozwiązania problemu, można przedstawić za pomocą drzewa (Rysunek 3.2), które jednak w przypadku implementacji algorytmu występuje niejawnie.

Rysunek 3.2 Drzewo przestrzeni stanów dla omawianego problemu



Źródło: opracowanie własne

3.2. Obliczenie wstępnego ograniczenia górnego

Zgodnie z opisem implementacji algorytmu B&B **przedstawionym wcześniej (2.2)**, obliczamy na początku ograniczenie górne dla problemu w sposób zachłanny. Przechodząc po odpowiednich wierszach macierzy i wiedząc, które wierzchołki nie zostały jeszcze odwiedzone, wyznaczamy szukaną wartość. Dla przedstawionej instancji problemu:

Rysunek 3.3

$$ograniczenie\ górne\ =\ 2\ (0\ \rightarrow\ 1)\ +\ 4\ (1 \rightarrow\ 3) +\ 1(3\ \rightarrow\ 2) +\ 5\ (2\ \rightarrow\ 0) =\ \mathbf{12}$$

Źródło: opracowanie własne

W nawiasach zaznaczono, które krawędzie zostały wybrane.

3.3. Redukcja macierzy – obliczanie ograniczenia dolnego dla korzenia

Znając postać macierzy kosztów dla korzenia, może być ona zredukowana, dzięki czemu obliczona zostanie dolna granica dla korzenia – jest to najmniejsza możliwa wartość kosztu rozwiązania. Wszystkie możliwe rozwiązania mogą być co najwyżej równe tej wartości, ale nie mniejsze. Redukcja przebiega zgodnie z algorytmem przedstawionym w opisie implementacji (**2.6**). Po prawej stronie macierzy oraz pod nią przedstawiono koszt redukcji danego wiersza lub danej kolumny. Koszt redukcji jest sumą wszystkich tych wartości.

Rysunek 3.4

$$\begin{array}{cccc|c} -1 & 2 & 7 & 3 & [-2] \\ 1 & -1 & 6 & 4 & [-1] \\ 5 & 2 & -1 & 8 & [-2] \\ 6 & 3 & 1 & -1 & [-1] \\ \hline - & - & - & [-1] & \end{array}$$

$$suma\ =\ 2\ +\ 1\ +\ 2 +\ 1 +\ 1\ =\ 7$$

Źródło: opracowanie własne

Suma kosztów redukcji dla korzenia wynosi **7**. Zredukowana macierz przyjmuje następującą postać:

Rysunek 3.5

$$\begin{array}{cccc} -1 & 0 & 5 & 0 \\ 0 & -1 & 5 & 2 \\ 3 & 0 & -1 & 5 \\ 5 & 2 & 0 & -1 \end{array}$$

Źródło: opracowanie własne

Jak można łatwo zauważyć, w każdym wierszu i każdej kolumnie jest przynajmniej jedno zero. Korzeń drzewa w postaci z zredukowaną macierzą oraz obliczonym kosztem (**7**) jest umieszczany na stosie. Następnie analizowane są (podobnie jak dla innych węzłów) jego potomkowie.

3.4. Analiza kolejnych węzłów

Najpierw rozpoczynamy od potomka korzenia związanego z węzłem nr 1. Wykluczamy odpowiednie elementy macierzy w sposób opisany wcześniej oraz, tak jak dla korzenia, wyznaczamy koszt redukcji. Należy przypomnieć, że potomek „dziedziczy" macierz po swoim poprzedniku.

Rysunek 3.6

$$\begin{array}{ccccc} -1 & -1 & -1 & -1 & - \\ -1 & -1 & \mathbf{\not{5}\,3} & \mathbf{\not{2}\,0} & [-2] \\ \mathbf{\not{0}\,3} & -1 & -1 & \mathbf{\not{5}\,2} & [-3] \\ 5 & -1 & 0 & -1 & - \\ - & - & - & [-1] & \end{array}$$

Źródło: opracowanie własne

Koszt redukcji wynosi 12, gdyż jest to suma obecnej redukcji oraz koszt poprzedniego węzła (7 + 5). Umieszczamy potomka na stosie, gdyż jego koszt (ograniczenie dolne) nie przekracza ograniczenia górnego. Podobne obliczenia należy wykonać dla węzłów oznaczonych indeksami 2 i 3. Dla nich koszty wynoszą odpowiednio 12 i 7 – ograniczenie górne nie jest przekroczone także w ich przypadku. Jako, że ostatni na stosie został umieszczony węzeł z indeksem 3, w pierwszej kolejności zdejmujemy właśnie go i rozpatrujemy jego potomków.

Rozpatrując przejście od 3 do 1 i wyznaczając dolne ograniczenie potomka 1, zauważamy, że koszt redukcji wynosi 15 (7 + 8, 8 to koszt redukcji macierzy tego konkretnego węzła). Jest to wartość większa od ograniczenia górnego – usuwamy węzeł i nie analizujemy go dalej. Przechodzimy do potomka z indeksem 2. Jego koszt wynosi 7, więc umieszczamy go na stosie. W następnej iteracji analizujemy jego jedynego potomka z indeksem nr 1 – jego koszt także wynosi **7**. Jest to jednocześnie liść, a więc możliwe rozwiązanie problemu. Jego koszt jest niższy od dotychczasowego ograniczenia górnego, więc je **korygujemy**. Zapisujemy także nową najlepszą ścieżkę (**0 →3 →2 →1**).

Zdejmując omówiony liść drzewa ze stosu, powracamy do analizowania potomków korzenia z indeksami 1 i 2. W przypadku każdego z nich zauważamy, że ich koszty (12) przekraczają nowe ograniczenie górne – nie mogą prowadzić one do lepszych rozwiązań. Dlatego je usuwamy. Stos staje się pusty i następuje koniec algorytmu. Rozwiązaniem jest przedstawiona wcześniej ścieżka o koszcie równym **7**.

4. Omówienie techniki programowania dynamicznego

W sprawozdaniu tym zostanie także omówione, z teoretycznego punktu widzenia, możliwe zastosowanie techniki programowania dynamicznego (ang. *dynamic programming*) w celu rozwiązywania problemu ATSP.

W przypadku programowania dynamicznego stosowana jest tabelaryczna metoda rozwiązywania problemów. Główny problem dzielony jest na mniejsze podproblemy, które jednak, w przeciwieństwie do podproblemów w metodzie „dziel i zwyciężaj", nie są niezależne[5]. W algorytmie programowania dynamicznego rozwiązujemy każdy podproblem tylko raz, a następnie zapisujemy związany z nim rezultat w tabeli. Rozwiązanie problemu końcowego wynika ze złożenia rozwiązań podproblemów.

4.1. Algorytm oparty na programowaniu dynamicznym dla problemu ATSP

Niech będzie dany (podobnie jak w przypadku metody B&B) ciąg wierzchołków grafu numerowany od 0, na przykład $\{0, 1, 2, 3 \ldots, N-1\}$, gdzie N to rozmiar problemu. Rozważmy wierzchołek 0 jako punkt początkowy i końcowy ścieżki. Dla każdego innego wierzchołka w różnego od 0 znajdujemy najkrótszą ścieżkę z wierzchołka 0 do wierzchołka *w*, która przechodzi przez wszystkie wierzchołki dokładnie raz. Niech koszt tej ścieżki wynosi *koszt(i)*, a koszt odpowiadającego cyklu wyniesie *koszt(i) + odl(i, 0)*, gdzie *odl(i, 0)* to odległość od *I* do 0. Ostatecznie zwracamy minimum spośród wszystkich wartości *[koszt(i) + odl(i, 0)]*.

Zdefiniujmy termin *K(S, i)* jako koszt ścieżki o minimalnym koszcie, odwiedzającej każdy wierzchołek w zbiorze S dokładnie raz, zaczynając od 0 i kończąc na *i*. Rozpoczynamy od wszystkich podzbiorów o rozmiarze 2 i obliczamy *K(S, i)* dla wszystkich podzbiorów, gdzie *S* to podzbiór. Następnie obliczamy *K(S, i)* dla wszystkich podzbiorów *S* o rozmiarze 3 - i tak dalej (Rysunek 4.1).

Rysunek 4.1 Algorytm obliczania K(S, i)

```
Jeżeli rozmiar podzbioru S wynosi 2, wtedy S ma postać {0, i}:
    K(S, i) = odl(0, i)

W przeciwnym wypadku – jeżeli rozmiar S jest większy niż 2:
    K(S, i) = min{ K(S-{i}, j) + odl(j, i)}, gdzie wierzchołek j
    należy do S oraz j != i oraz j!=0
```

Źródło: opracowanie własne na podstawie (7)

W celu oznaczania wierzchołków, które są odwiedzane w danym podzbiorze, można użyć manipulacji na reprezentacjach bitowych liczb, co jest metodą szybką pod względem wydajnościowym[6].Złożoność czasowa algorytmu zaimplementowanego w przedstawiony sposób wynosi **$O(N^2 * 2^N)$.** [7]

[5] http://www.cs.put.poznan.pl/arybarczyk/TeoriaAiSD3.pdf
[6] https://www.geeksforgeeks.org/travelling-salesman-problem-using-dynamic-programming/
[7] https://www.baeldung.com/cs/tsp-dynamic-programming

5. Sposób przeprowadzenia eksperymentu

W celu realizacji badania – eksperymentu zostały wykorzystane omówione wcześniej klasy. Podobnie jak w przypadku realizacji projektu nr 1, w celu oceny efektywności badanych algorytmów został wykorzystany pomiar czasu przy wykorzystaniu funkcjonalności biblioteki <*chrono*> i zawartej w niej klasy *steady_clock*, która reprezentuje zegar monotoniczny, dla którego gwarantowane jest, że różnica czasu (przykładowo przekonwertowanego do milisekund – tak jak w programie) będzie większa od zera dla dwóch momentów czasu badania, z których drugi występuje później niż pierwszy. Różnica czasów pobranych przed i po wykonaniu algorytmu jest zwracana z metody.

W głównym programie, z którego korzysta użytkownik, możliwe jest wywołanie metody dla pojedynczej instancji problemu o rozmiarze *N*, która może być wylosowana lub wczytana z pliku, oraz przeprowadzenie zbiorczego badania dla 100 losowych instancji problemu o rozmiarze *N*, dzięki czemu możliwe jest znalezienie średniego czasu działania algorytmu jako iloraz sumy czasów poszczególnych realizacji algorytmu i ich ilości. Należy zaznaczyć, że obie opcje mogą zostać zrealizowane z wykorzystaniem metody z lub bez limitu czasu wykonywania algorytmu. Opcja badania zbiorczego została wykorzystana do przeprowadzenia analizy efektywności algorytmu rozwiązującego problem ATSP, która jest przedmiotem tego projektu. Porównano omówiony w tym projekcie algorytm realizujący metodę B&B z algorytmem przeglądu zupełnego omówionym w projekcie nr 1. W tym celu oba algorytmy przetestowano dla tych samych instancji problemu o określonym rozmiarze *N*. Badania przeprowadzono dla następujących wartości *N*: **9, 11, 12, 13, 16, 19, 21, 23, 25, 28, 32**. Ustalono maksymalny czas wykonywania każdego algorytmu na **1 minutę**.

Badania zostały przeprowadzone na laptopie z systemem operacyjnym Windows 10 Home, procesorem Intel Core i7-8750H, 24 GB pamięci RAM oraz dyskiem SSD.

6. Wyniki przeprowadzonego eksperymentu

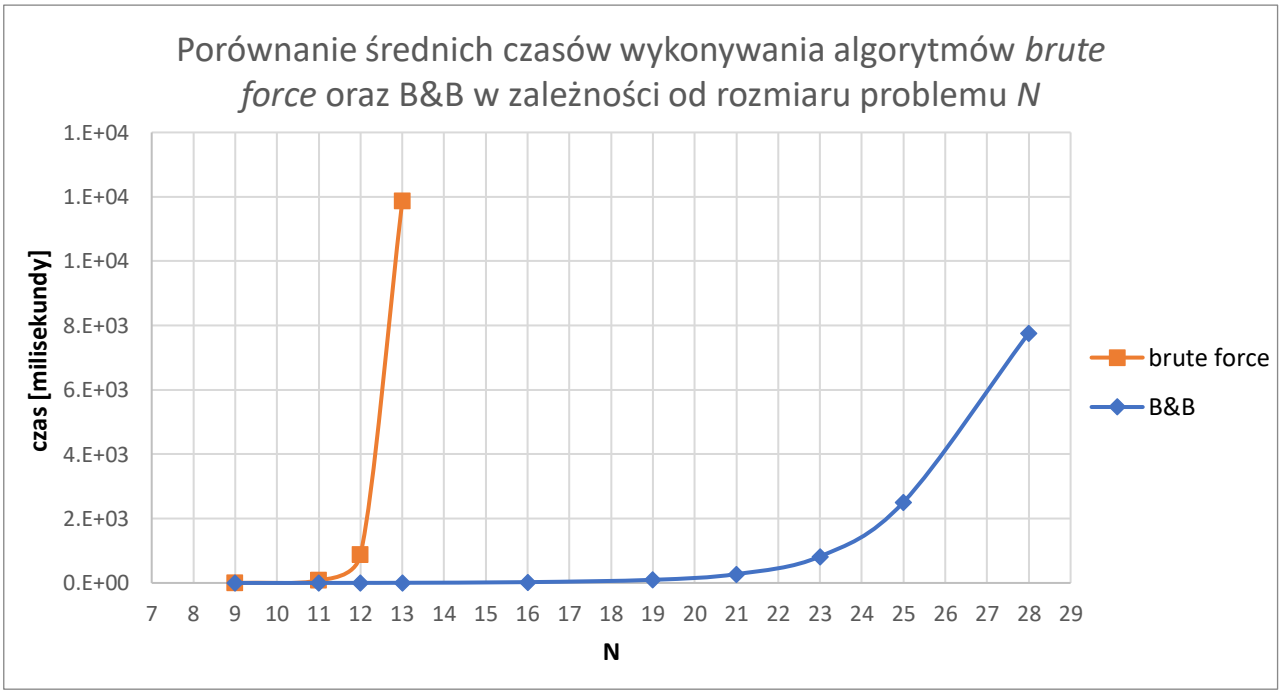
Rezultaty przeprowadzonego badania przedstawione są w tabeli oraz na wykresach. Limit czasu dla algorytmu *brute force* został przekroczony dla **wartości *N* większych od 13**. Dla algorytmu B&B liczba przerwanych realizacji algorytmu **przekroczyła 20% dla *N* = 32**, w związku z czym zaprzestano dalszej analizy.

Tabela 6.1 Średnie czasy realizacji poszczególnych algorytmów dla danych rozmiarów problemów N

	Algorytm	
N	**przegląd zupełny (*brute force*) [ms]**	**B&B [ms]**
9	1	0
11	81	1
12	880	2
13	$1{,}19 \times 10^4$	5
16	-	25
19	-	97
21	-	275
23	-	823
25	-	$2{,}51 \times 10^3$
28	-	$7{,}77 \times 10^3$
32	-	-

Źródło: opracowanie własne

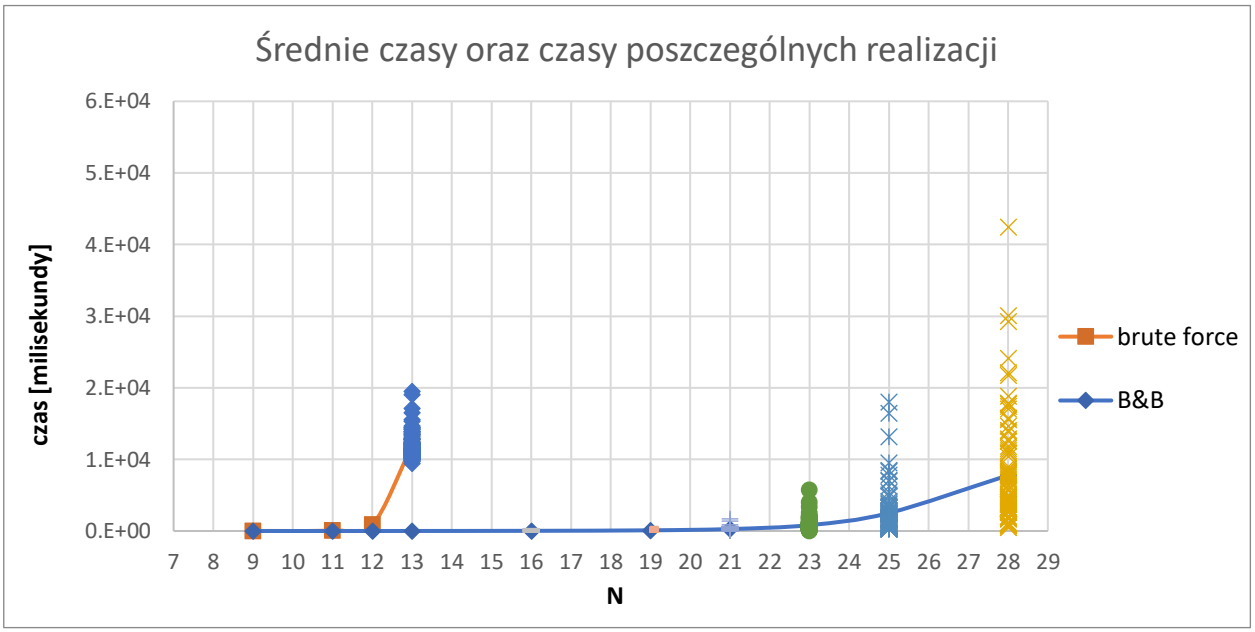
Wykres 6.1 Średnie czasy wykonywania algorytmów



Źródło: opracowanie własne

Ponadto należy nadmienić, że w przypadku algorytmu B&B dopiero dla *N*=28 wystąpiła niezerowa liczba przerwań algorytmu - 3% realizacji zakończyło się jego przerwaniem – przekroczono limit **1 minuty**.

Wykres 6.2 Ten sam wykres, co Wykres 6.1, ale z uwzględnieniem czasów poszczególnych realizacji



Źródło: opracowanie własne

Wykres 6.2 w przeciwieństwie do Wykres 6.1 oprócz średnich wartości czasu przedstawia także czasy poszczególnych realizacji algorytmu. Można zaobserwować, że w przypadku tego wykresu czasy realizacji algorytmu B&B dla problemów o większych rozmiarach *N* mają decydujący wpływ na czytelność wykresu ze względu na bardzo duże odchylenie tych czasów od wartości średnich.

7. Wnioski

Można wywnioskować, że omówiony algorytm realizujący metodę podziału i ograniczeń (B&B) jest z pewnością algorytmem trudniejszym do zrozumienia i zaimplementowania od algorytmu przeglądu zupełnego, jednak jest on zdecydowanie efektywniejszy pod względem obliczeniowym. Różnica czasów wykonywania obu algorytmów dla tych samych instancji problemu okazała się znacząca. Należy jednak zwrócić uwagę na fakt, że w przypadku algorytmu B&B duże znaczenie na czas wykonywania algorytmu ma, oprócz rozmiaru problemu, jego „stopień trudności", co ma potwierdzenie w różnicach w czasach rozwiązywania problemów o tym samym rozmiarze. Jest to potwierdzeniem faktu, że trudno oszacować złożoność obliczeniową dla algorytmów tego typu. Udało się poprawnie zaimplementować algorytm realizujący metodę B&B, zbadać jego efektywność i porównać ją z efektywnością algorytmu *brute force*.

8. Bibliografia

[1] B&B (1) - https://ii.uni.wroc.pl/~prz/2011lato/ah/opracowania/met_podz_ogr.opr.pdf
[2] B&B (2) - https://eugeniucozac.medium.com/what-is-a-branch-and-bound-algorithm-925aeecef0b1