
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.....	1
2. OPIS IMPLEMENTACJI ALGORYTMU	2
2.1. KLASA <i>GRAPH</i>	2
2.2. KLASA <i>GRAPH</i> - OBLICZENIE OGRANICZENIA GÓRNEGO (<i>UPPER BOUND</i>)	2
2.3. KLASA <i>GRAPH</i> – METODA <i>TIMEBRANCHANDBOUNDATSP()</i>	3
2.4. KLASA <i>GRAPH</i> – METODA <i>TIMEBRANCHANDBOUNDATSPLIMITED()</i>	4
2.5. KLASA <i>BNBNODE</i>	4
2.6. KLASA <i>BNBNODE</i> – METODA <i>REDUCEMATRIX()</i>	4
2.7. KLASA <i>BNBSTACK</i>	5
3. PRZYKŁAD PRAKTYCZNY DZIAŁANIA ALGORYTMU „KROK PO KROKU”	6
3.1. REPREZENTACJA DRZEWA PRZESTRZENI STANÓW DLA PROBLEMU	7
3.2. OBLICZENIE WSTĘPNEGO OGRANICZENIA GÓRNEGO	8
3.3. REDUKCJA MACIERZY – OBLICZANIE OGRANICZENIA DOLNEGO DLA KORZENIA	8
3.4. ANALIZA KOLEJNYCH WĘZŁÓW	9

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 wszerek) - 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.

¹ https://en.wikipedia.org/wiki/Branch_and_bound

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 wężła 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 wężła znajdujemy minimum w każdym wierszu – wartość w kolumnie o indeksie wężła jeszcze nieodwiedzzonego (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 // Dopóki 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     Jeżeli wierzchołek to liść i jego koszt jest mniejszy od górnej granicy,
126     zapisujemy nową najlepszą ścieżkę i górną granicę. Dealokujemy pamięć. Kontynuujemy algorytm od kolejnego
127     wierzchołka 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 wierzchołka
138     int currentNode = current->getNode();
139
140     /*
141     Iterujemy po wszystkich możliwych potomkach danego węzła. Jeżeli jego koszt jest mniejszy niż wartość
142     górnego ograniczenia (jest "obiecujący"), 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 pamięć po obecnym wierzchołku
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 wykonanń 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.

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 odwiedzionym 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 macierzy² - macierz zredukowana jest w taki sposób, aby w każdym wierszu i każdej kolumnie znalazło się przynajmniej jedno zero, a całkowity koszt redukcji jest zwracany z metody.

Przechodząc po wszystkich wierszach i kolumnach, znajdujemy dla każdego wiersza i kolumny **nieujemne** minimum, które następnie odejmujemy od danego wiersza/kolumny. Suma tych

² <https://cs.pwr.edu.pl/zielinski/lectures/om/mow10.pdf>

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 jednocześnie po wierszach i kolumnach macierzy wierzchołka
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 $N = 4$:

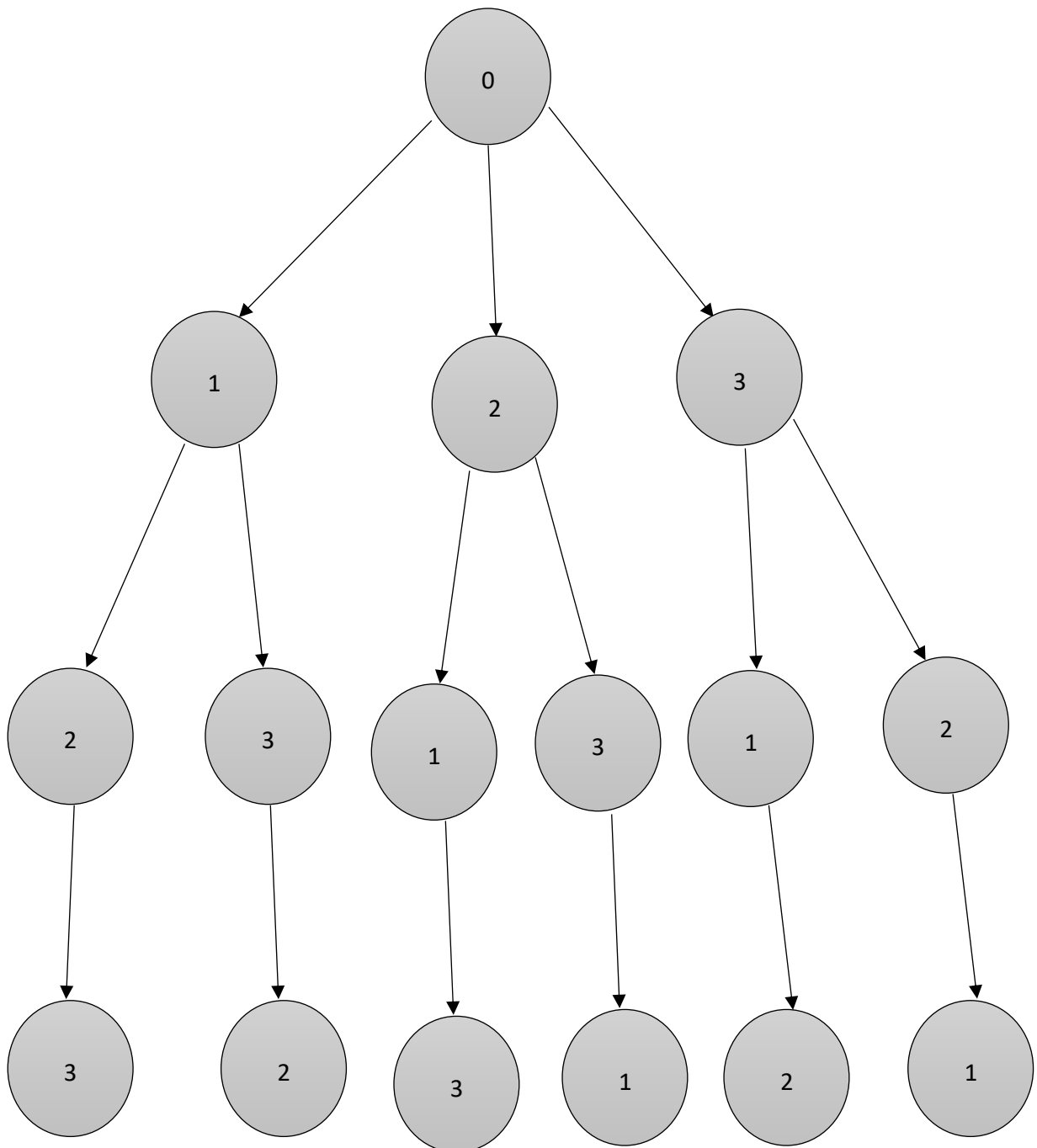
	0	1	2	3
0	–1	2	7	3
1	1	–1	6	4
2	5	2	–1	8
3	6	3	1	–1

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 3.1), które jednak w przypadku implementacji algorytmu występuje niejawnie.

Rysunek 3.1



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:

$$\text{ograniczenie g\acute{o}rne} = 2 (0 \rightarrow 1) + 4 (1 \rightarrow 3) + 1 (3 \rightarrow 2) + 5 (2 \rightarrow 0) = \mathbf{12}$$

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.

-1	2	7	3	[-2]
1	-1	6	4	[-1]
5	2	-1	8	[-2]
6	3	1	-1	[-1]
-	-	-	[-1]	

$$\text{suma} = 2 + 1 + 2 + 1 + 1 = 7$$

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

-1	0	5	0
0	-1	5	2
3	0	-1	5
5	2	0	-1

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.

$$\begin{array}{ccccc} -1 & -1 & -1 & -1 & - \\ -1 & -1 & 5 & 3 & 2 \quad 0 \quad [-2] \\ 0 & 3 & -1 & -1 & 5 & 2 \quad [-3] \\ 5 & -1 & 0 & -1 & - \\ - & - & - & [-1] \end{array}$$

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 jedyne 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.