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].

[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 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 wykonania każdej 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.

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.

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

Metoda ta jest wykorzystana do obliczenia ograniczenia dolnego poprzez algorytm redukcji macierzy[2] - 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.

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

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

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*. 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.