Projektowanie Efektywnych Algorytmów

Zadanie nr 1 – Asymetryczny Problem Komiwojażera: Brute Force, Branch And Bound, Dynamic Programming

Wykonawca: Marcel Guzik 256317
Prowadzący: dr inż Jarosław Mierzwa
Grupa: Piątek 15:15 grupa późniejsza
Data realizacji zadania: 03.12.2021

Wstęp

Celem zadania jest rozwiązanie asymtrycznego problemu komiwojażera za pomocą trzech metod: przeglądu zupełnego (brute force), podziału i ograniczeń (branch and bound) oraz programowania dynamicznego (dynamic programming).

Brute force

Metoda Brute Force polega na wygenerowaniu każdej możliwej ścieżki (czyli permutacji) a następnie wybraniu najkrótszej. Jest to algorytm najprostszy do zrealizowania, lecz wysoce nieoptymalny. Dzięki użyciu innych metod jesteśmy w stanie znacząco ograniczyć przestrzeń do przeszukania. Oprócz prostoty, jedną z niewielu zalet algorytmu jest także zużycie pamięci – algorytm w każdej chwili pamięta tylko jedną permutację, ponieważ tyle wystarczy do wygenerowania następnej.

Złożoności: obliczeniowa – $O(n!)$; pamięciowa - $O(1)$

Branch and Bound

Metoda branch and bound polega na przechowywaniu bieżącego stanu przechowywania jako drzewo, a następnie eksplorowanie wybranych gałęzi drzewa, najczęściej tych dla których przewidywany wynik będzie najniższy. Ponadto, jeżeli algorytm znajdzie jakieś rozwiązanie, wyznacza ono górną granicę rozwiązania, a wszystkie gałęzie dla których przewidywany koszt jest większy niż owa granica, zostają usunięte. Rozpoczyna ona wyszukiwanie od miasta startowego i metodą depth first search rozwija każdy wierzchołek drzewa.

Metoda Branch and Bound kosztem większego zużycia pamięci, jest w stanie znacząco ograniczyć przestrzeń wyszukiwania i tym samym ilość operacji potrzebnych do wykonania.

Złożoności: obliczeniowa – najgorsza $O(n!)$, najlepsza $O(n)$; pamięciowa: najlepsza $O(n)$, najgorsza $O(n!)$

Dynamic Programming

Ogólnie rzecz biorąc, programowanie dynamiczne to strategia projektywania algorytmów polegająca na rozbiciu dużego problemu na zbiór mniejszych, nakładających się subproblemów.

W wypadku problemu komiwojażera, metoda programowania dynamicznego, a dokładniej rzecz biorąc algorytm Helda-Karpa, polega na obliczeniu najkrótszej ścieżki do każdego z miast – najpierw bezpośrednio, następnie przez jedno miasto, następnie przez 2, etc. aż do momentu kiedy do miasta dochodzimy odwiedzając po drodze wszystkie inne miastaw określonej kolejności.

Aby uzyskać koszt dojścia do miasta **f** przez zbiór miast o wielkości **i + 1**, należy dla każdego miasta **c != f, c_0**, wybrać najkrótszą ścieżkę od miasta startowego do miasta **c** i do tej ścieżki dodać krawędź **c**→**f**. Spośród zbioru uzyskanych w ten sposób ścieżek wybieramy najkrótszą. W ten sposób, zbiór ścieżek o długości **i** jesteśmy w stanie przekształcić w zbiór ścieżek o długości **i + 1**, aż do momentu w którym **i = n**, kiedy to przechodzimy przez każdy wierzchołek dokładnie jeden raz, możemy wtedy dla każdej z finalnych ścieżek od miasta c_0 do miasta c_f dołączyć krawędź $c_f \rightarrow c_0$, uzyskując finalne rozwiązanie.

Złożoności: Obliczeniowa $O(n^2 2^n)$, pamięciowa $O(n 2^n)$.

Testowanie

Program kompilowano i testowano na systemie o następującej specyfikacji:

CPU: Ryzen 5 3600 6c/12t 3.6GHz 384KB/3MB/32MB cache
OS: Linux 5.14.21-2-MANJARO
glibc: 2.23
skompilowano: `g++ -std=c++20 -g -O -Wall -Wextra -Wpedantic src/main.cpp -o zad1.out`

Dla każdego algorytmu generowane są instancje problemu – po 100 dla każdego rozmiaru instancji od 12 do 20. Instancje są generowane i zapisywane, a następnie mierzony jest czas wykonania wszystkich 100 instancji dla danego rozmiaru N.

Wyniki

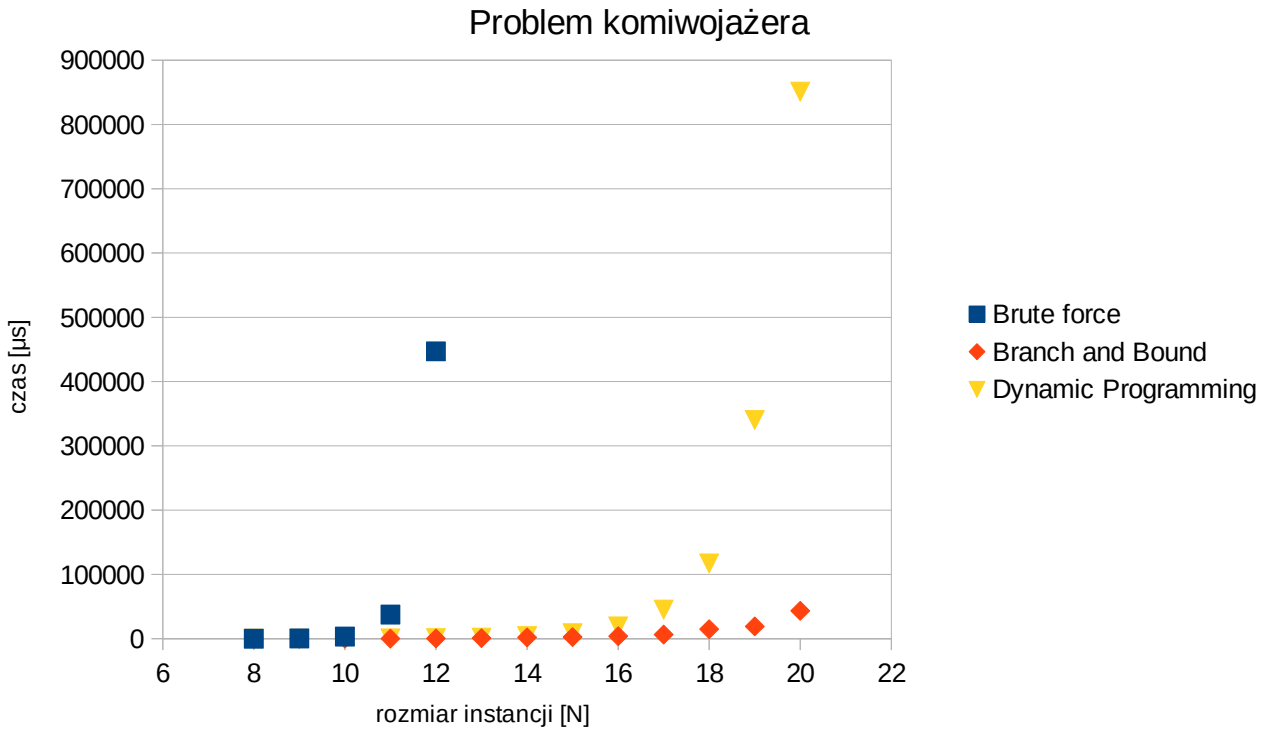

Figure 1: Wyniki testu dla N od 8 do 20. Skala liniowa.

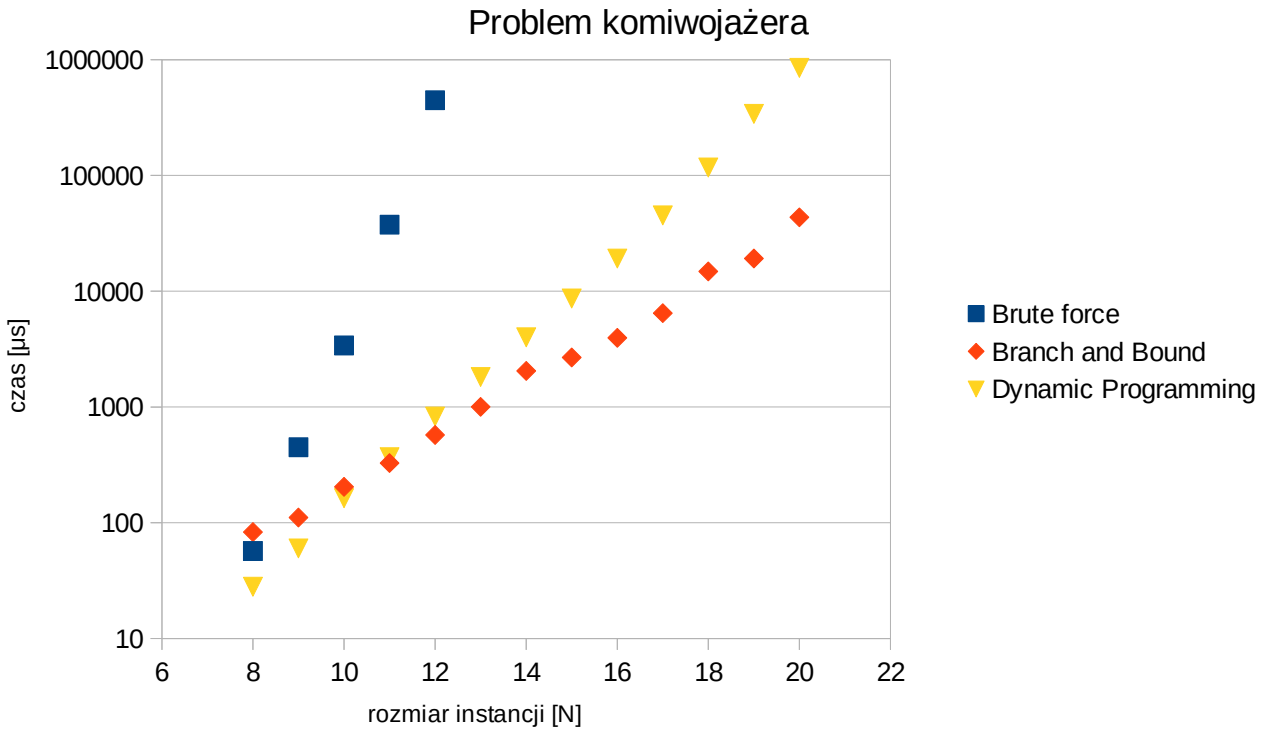

Figure 2: Wyniki testu dla N od 8 do 20. Skala logarytmiczna.

Wyniki pomiarów potwierdzają przewidywaną niską wydajność algorytmu Brute Force. Zaskakująca jest natomiast o wiele niższa wydajność algorytmu Helda-Karpa w porównaniu do metody branch and bound.

Analiza niskiej wydajności algorytmu Helda-Karpa

Do zbadania możliwych przyczyn niskiej wydajności algorytmu Helda-Karpa przygotowano wersję programu wykonującą 10 powtórzeń algorytmu Helda-Karpa, a następnie branch and bound dla N=20 oraz wykorzystano narzędzie `perf stat`.

Held-Karp:

```
perf stat ./zad1.out

 Performance counter stats for './zad1.out':

         76 514,38 msec task-clock:u              #    1,000 CPUs utilized
                 0      context-switches:u        #    0,000 /sec
                 0      cpu-migrations:u          #    0,000 /sec
         2 279 851      page-faults:u             #   29,796 K/sec
   304 942 251 718      cycles:u                  #    3,985 GHz                      (83,33%)
     8 373 733 684      stalled-cycles-frontend:u #    2,75% frontend cycles idle     (83,33%)
   142 482 864 124      stalled-cycles-backend:u  #   46,72% backend cycles idle      (83,33%)
   270 899 439 695      instructions:u            #    0,89  insn per cycle
                                                  #    0,53  stalled cycles per insn  (83,34%)
    64 511 432 731      branches:u                #  843,128 M/sec                    (83,33%)
     3 672 048 640      branch-misses:u           #    5,69% of all branches          (83,33%)

      76,535140450 seconds time elapsed

      73,822736000 seconds user
       2,207504000 seconds sys
```

Branch and bound:

```
Performance counter stats for './zad1.out':

          3 270,24 msec task-clock:u              #    1,000 CPUs utilized
                 0      context-switches:u        #    0,000 /sec
                 0      cpu-migrations:u          #    0,000 /sec
            45 024      page-faults:u             #   13,768 K/sec
    13 207 817 278      cycles:u                  #    4,039 GHz                      (83,31%)
       329 137 917      stalled-cycles-frontend:u #    2,49% frontend cycles idle     (83,31%)
       805 348 617      stalled-cycles-backend:u  #    6,10% backend cycles idle      (83,31%)
    21 817 360 787      instructions:u            #    1,65  insn per cycle
                                                  #    0,04  stalled cycles per insn  (83,33%)
     5 114 384 457      branches:u                #    1,564 G/sec                    (83,40%)
       220 483 079      branch-misses:u           #    4,31% of all branches          (83,36%)

       3,271071621 seconds time elapsed

       3,204542000 seconds user
       0,046427000 seconds sys
```

Należy zauważyć wysoką w porównaniu do Branch and Bound wartość stalled-cycles-backend, która oznacza iż dane nie trafiają do backendu procesora wystarczająco szybko. Powodem najczęściej jest niski cache hit rate. Na podstawie tych wyników sformułowano następującą hipotezę:

Objętość pamięciowa algorytmu Helda-Karpa jest o wiele większa niż pamięć cache procesora, a wzory dostępu do pamięci powodują wysoki cache miss rate.

Złożoność pamięciowa algorytmu Helda-Karpa wynosi $O(n2^n)$. W niniejszej jego implementacji, jedynym znaczącym elementem pod względem pamięci jest tablica `distances`. Przechowuje ona dla każdego wierzchołka wszystkie częściowe ścieżki kończące się w tym

wierzchołku. Zbiór miast przez który przechodzi ścieżka jest przechowywany jako int, a więc 4 bajty. Tablica ta ma n * 2^n elementów. Przykładowo: dla N=20 rozmiar tej tablicy wynosi 20 * 2^20 * 4B = 80MB. Dla tylko 4 miast więcej, a więc N=24, rozmiar tablicy wynosi 24 * 2^24 = 1,5GB.

Aby zweryfikować tą hipotezę, zmierzono wartość stalled-cycles-backend dla wykonania po jednym powtórzeniu problemu dla każdej z wielkości instancji od 10 do 20 dla obu algorytmów. Jeżeli hipoteza jest prawdziwa, wartość stalled-cycles-backend powinna rosnąć w czasie dla wysokich rozmiarów instancji dla algorytmu Helda-Karpa, a pozostać relatywnie stała dla algorytmu Branch and Bound.

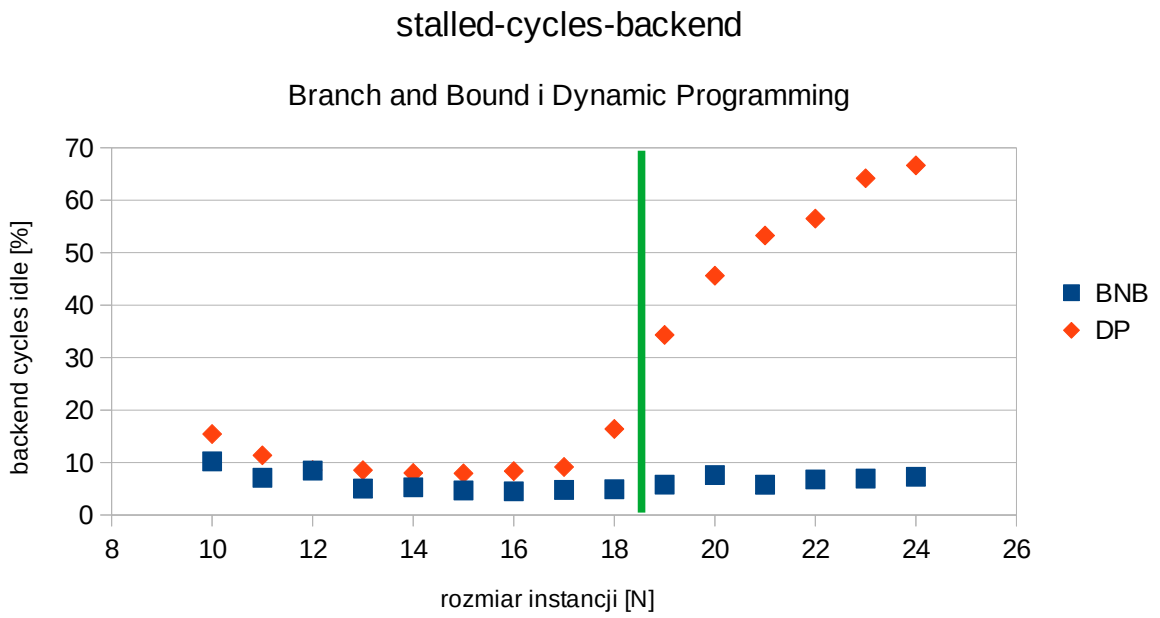

Figure 3: Wykres prezentujący zależność wartości backend cycles idle od wielkości instancji. Zieloną linią oznaczono projektowany rozmiar instancji której objętość pamięciowa byłaby równa wielkości pamięci L3 cache procesora.

Wykonany wykres zdaje się potwierdzać hipotezę. Jasno widoczna jest tendencja wzrostowa dla wyższych instancji dla algorytmu Helda-Karpa oraz stały, < 10% backend stall dla algorytmu Branch and Bound. Ponadto zaznaczono na wykresie interpolację rozmiaru instancji dla którego wykonanie algorytmu zużywałoby ilość pamięć równą ilości pamięci cache procesora.

Wnioski

Najlepszym algorytmem okazał się algorytm Branch and Bound. Algorytm Helda-Karpa, choć teoretycznie czasowo powinien być bliski Branch and Bound, okazał się być wolniejszy z powodu użycia zbyt dużej ilości pamięci oraz nieoptymalnymi wzorami dostępu do niej. Nie jest wiadome czy można wyeliminować niekorzystne wzorce dostępowe i polepszyć wkaźnik trafień w pamięć podręczną procesora.

Zadanie wykazało, iż oprócz teoretycznej ewaluacji algorytmu, należy przede wszystkim rozważyć charakterystykę sprzętu na którym będzie on wykonywany.