Projektowanie Efektywnych Algorytmów

Projekt

08/11/2021

264067 Mateusz Waszczuk

(3) Breach and Bound

1.Sformułowanie zadania

Zadanie polega na opracowaniu, implementacji i zbadaniu efektywności algorytmu Breach and Bound rozwiązującego problem komiwojażera w wersji optymalizacyjnej. Oznacza to że celem niniejszego zadania jest stworzenie programu który znajdzie wszystkie możliwe cykle Hamiltona w zadanych grafach oraz sprawdzi który ze znalezionych cykli ma najmniejszy koszt, tzn. sumę wag tworzących go krawędzi. Program ma mierzyć czas wykonania algorytmu breach and bound dla każdej instancji. Dane wejściowe dotyczące ilości powtórzeń dla każdej instancji oraz poprawne minimalne cykle Hamiltona wraz z właściwymi kosztami zostaną umieszczone w pliku inicjującym pomiary.ini. Minimalne cykle Hamiltona i koszty mają pełnić jedynie funkcję weryfikacji poprawności działania algorytmu, aby mieć pewność, że pomiary czasu będą miarodajne. Wyniki mają zostać zwrócone w pliku pomiary.csv. Po zaimplementowaniu algorytmu breach and bound następnym zadaniem będzie przeprowadzanie testów poprawności jego działania oraz określenie liczby pomiarów dla wybranych instancji na podstawie czasu pojedynczego wykonania, co zamierzam zrobić jednym plikiem inicjującym wstępne_testy.ini. Następnie należy przeprowadzić pomiary czasu dla odpowiednio skonfigurowanego już pliku inicjującego pomiary.ini oraz pomiary zużycia pamięci w zależności od wielkości instancji.

Celem niniejszego opracowania jest zbadanie zależności między czasem, a ilością wierzchołków w grafie oraz zależności między ilością wierzchołków a zużyciem pamięci podczas wykonywania zaimplementowanego przeze mnie algorytmu breach and bound używającego przeszukiwania grafu w głąb. Spodziewanym efektem jest otrzymanie znacznie większej wydajności niż w przypadku algorytmu brute force przy jednoczesnym utrzymaniu gwarancji optymalności rozwiązania. Spodziewana klasa złożoności implementowanego algorytmu powinna być mniejsza od O(n!) i większa niż O(1). Po otrzymaniu wykresu, jeśli któraś z instancji lub kilka instancji będzie wyraźnie odbiegać od trendu postaram się odpowiedzieć na pytanie czym jest to spowodowane.

2.Metoda

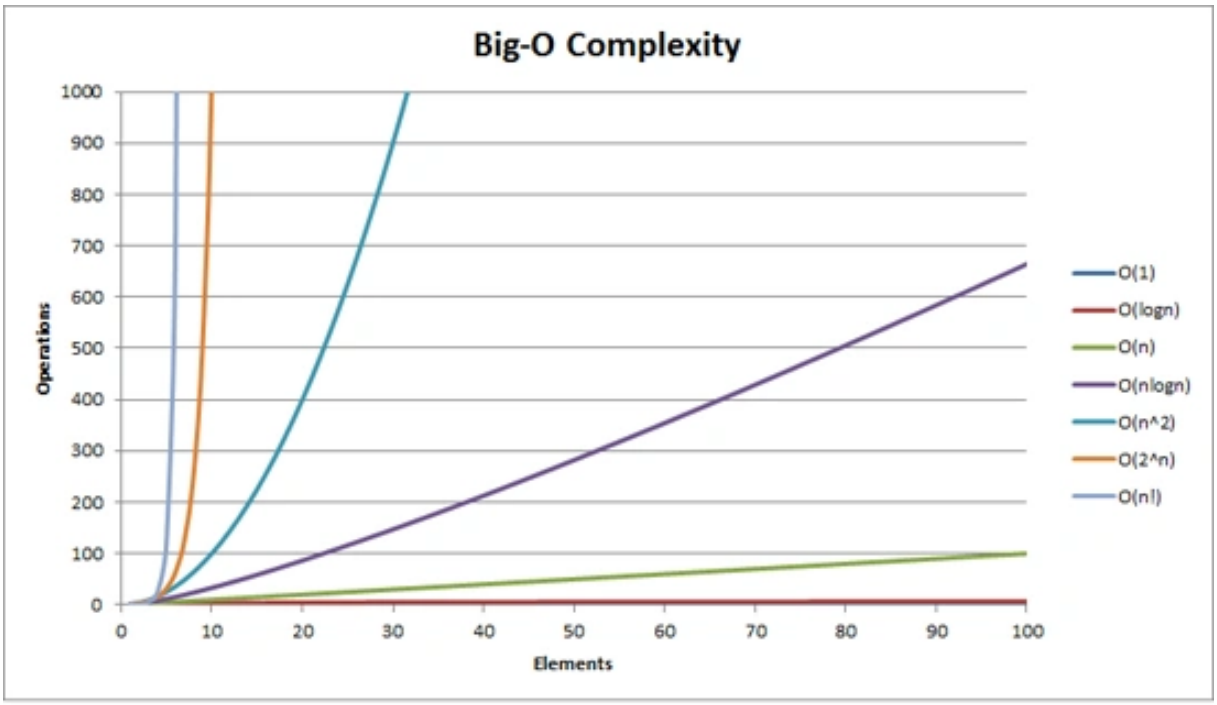
Metoda breach and bound, zwana także branch and bound, jest podobna do metody siłowej, w ten sposób, że tak samo jak metoda siłowa zwraca rozwiązanie optymalne, jednak zawiera mechanizmy które znaczenie ograniczają ilość koniecznych operacji, dzięki czemu jest bardziej wydajna czasowo. Metoda breach and bound może w celu znalezienia rozwiązania przeszukiwać graf różnymi metodami. Na potrzeby tego opracowania po krótce omówię trzy. Są to: metoda depth first search zwana dalej FS, breadth first search zwana dalej BFS oraz low cost, zwana dalej LC.

- Metoda DFS polega na eksploracji grafu poprzez przechodzenie jak najdalej wzdłuż jednej ścieżki (gałęzi) zanim zostaną odwiedzone pozostałe gałęzie. Algorytm rozpoczyna od wybranego wierzchołka źródłowego, a następnie przechodzi do dowolnego sąsiadującego wierzchołka, który nie został jeszcze odwiedzony. Algorytm kontynuuje eksplorację w głąb, aż osiągnie wierzchołek końcowy lub wierzchołek, który nie ma już nieodwiedzonych sąsiadów, a następnie cofa się o jeden krok i próbuje odwiedzić inne gałęzie, jeśli istnieją. Dla jej implementacji typowa jest rekurencja i mój algorytm używa właśnie podejścia rekurencyjnego.

- Metoda BFS jest również jednym z podstawowych algorytmów przeszukiwania grafu, ale różni się od DFS w podejściu do eksploracji. W metodzie BFS, algorytm rozpoczyna od wierzchołka źródłowego i odwiedza wszystkich jego sąsiadów, a następnie przechodzi do sąsiadów sąsiadów. Innymi słowy, ten algorytm przechodzi przez wierzchołki na jednym poziomie w grafie, zanim przejdzie do wierzchołków na poziomie następnym. Metoda BFS jest zazwyczaj realizowana za pomocą kolejki.

- Metoda LC każdorazowo wybiera wierzchołek o najniższym koszcie (minimalizacja) lub najwyższym koszcie (maksymalizacja) do odwiedzenia jako następny. Metoda ta nie jest więc związana z konkretnym podejściem przeszukiwania, ale z kryterium wyboru wierzchołka do odwiedzenia w danej chwili.

Algorytm, którego implementację wykonałem, jak już wspomniano, przeszukuje graf w głąb, używa w tym celu rekurencji i wyłącznie do tej implementacji odnosi się to opracowanie. Złożoność czasowa algorytmu breach and bound wynosi od O(1) do O(n!) (Rysunek 1). W najgorszym możliwym przypadku złożoność czasowa algorytmu breach and bound może być taka jak algorytmu brute force, a w najlepszym O(1).

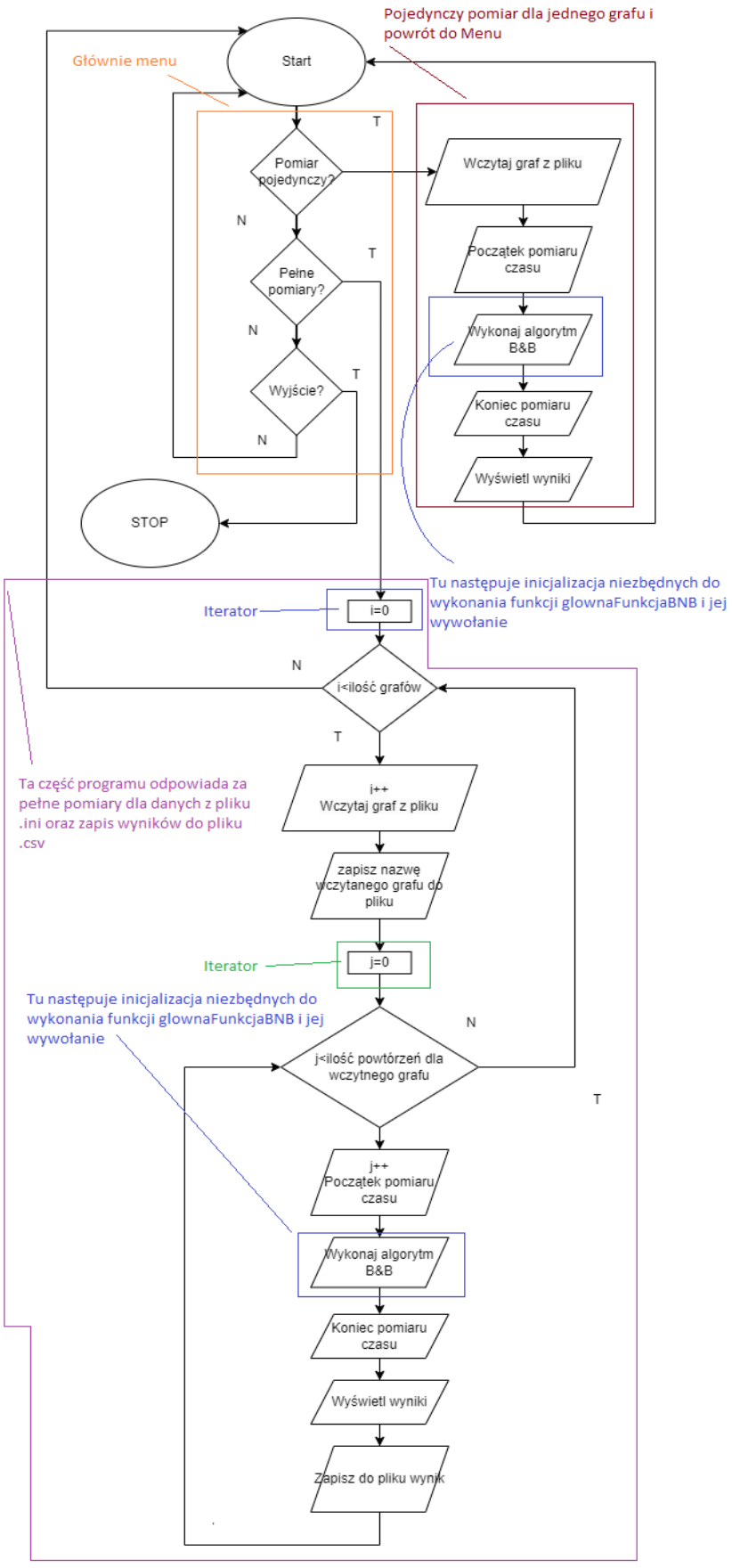


Rysunek 1: Wykres poglądowy porównujący różne klasy złożoności, w tym O(1) i O(n!)

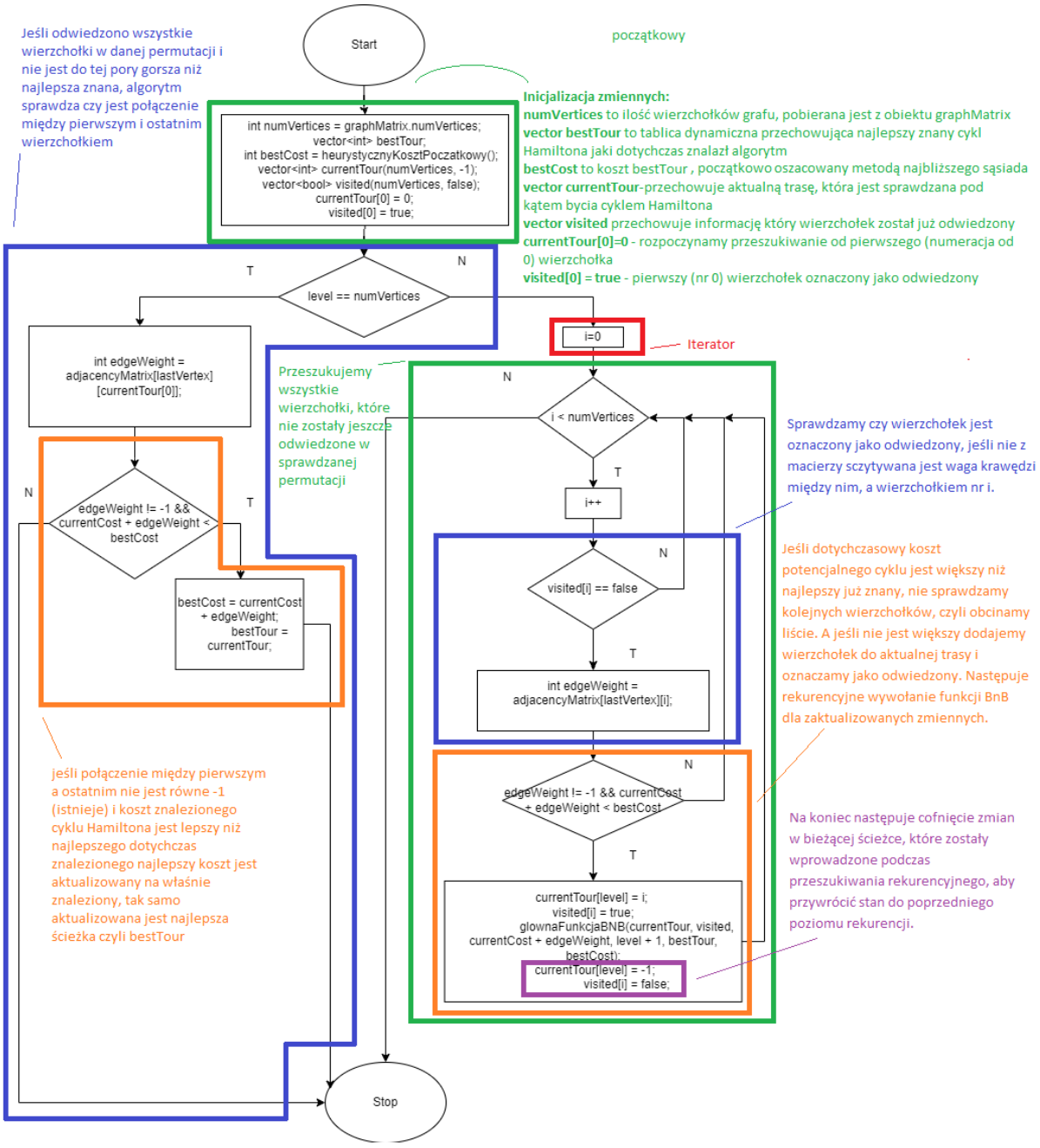
3.Algorytm

Na poniższym rysunku (Rysunek 2) znajduje się ogólny schemat blokowy całego programu który realizuje pomiary, zaś na Rysunku 3 znajduje się schemat blokowy części programu odpowiedzialnej za samą logikę algorytmu breach and bound. Co istotne, algorytm breach and bound większość operacji realizuje w metodzie glownaFunkcjaBNB. Metoda branchAndBound zawiera tylko inicjalizację niektórych zmiennych, a jej głównym zadaniem jest przeprowadzenie pomiarów czasu oraz wyświetlenie wyników. Schemat algorytmu breach and bound (Rysunek 3) obejmuje więc te operacje które są częścią algorytmu breach and bound, a nie stanowią wyłącznie określoną metodę. Analogiczne komentarze znajdują się w pliku Algorytmy.cpp który zawiera metodę bruteForce, a w niej sam algorytm brute force.

Istnieje też metoda wyznaczająca ograniczenie początkowej wartości najlepszego znanego kosztu heurystycznie za pomocą algorytmu najbliższego sąsiada. Nie jest ona jednak częścią samego algorytmu breach and bound, a jedynie ograniczeniem dla niego, można bowiem w celu ustalenia początkowej wartości najlepszego znanego kosztu użyć dowolną metodę heurystyczną, więc jej schemat nie został stworzony. Jej wywołanie jest jednak ujęte na schemacie blokowym metody breach and bound (Rysunek 3).



Rysunek 2: Ogólny schemat blokowy całego programu



Rysunek 3: Schemat blokowy algorytmu breach and bound

4.Dane testowe

Do sprawdzenia poprawności działania algorytmu oraz do określenia zestawu instancji do pomiarów użyto następujący zestaw instancji:

-tsp_10.txt, min. koszt: 212

-tsp_12.txt, min. koszt: 264

-tsp_14.txt, min. koszt: 282

-gr17.txt, min koszt 2085

-burma14.txt, min. koszt 3367

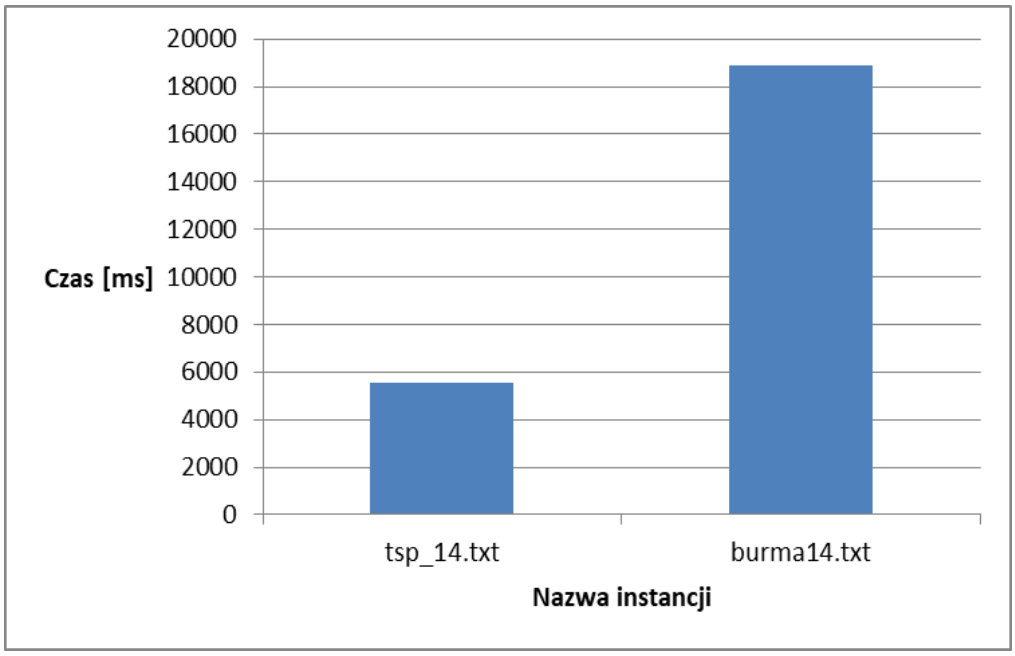
-tsp_17.txt, min. koszt 39

-gr_21.txt, min koszt 2707

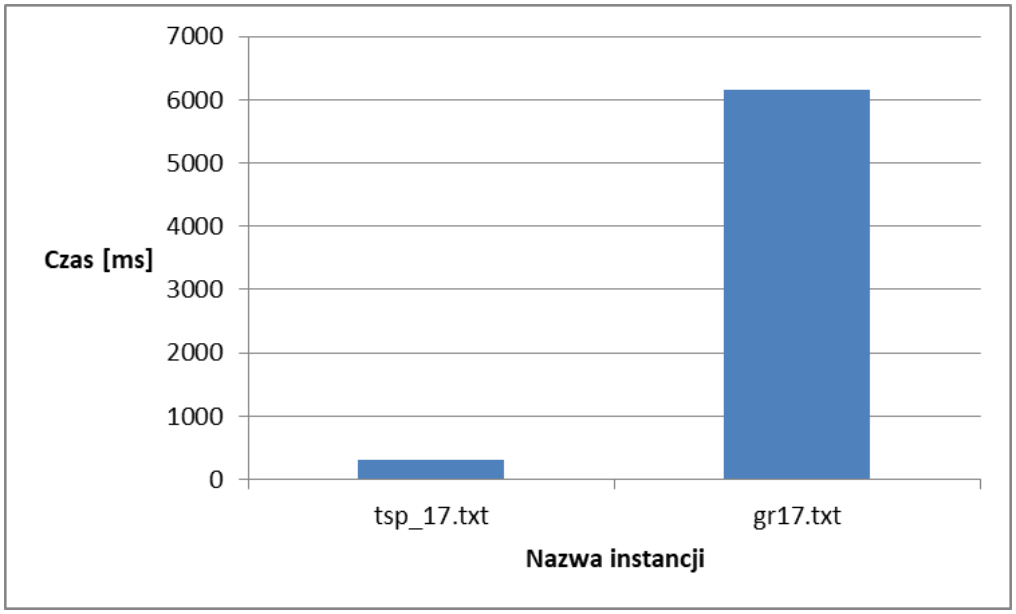
http://jaroslaw.mierzwa.staff.iiar.pwr.wroc.pl/pea-stud/tsp/

https://github.com/PrzemekRychter/PEA/tree/main/held_karp/instancjeDoZbadania

Po przetestowaniu programu za pomocą wszystkich ww. instancji koszty oraz optymalne ścieżki były zgodne z podanymi. Czas pojedynczego wykonania algorytmu dla gr_21 przekroczył 3 godziny, podjęto więc decyzję o zatrzymaniu programu. Okazało się także, że czas wykonania algorytmu dla pliku burma14.txt był kilkukrotnie wyższy niż dla pliku tsp_14.txt (Rysunek4) mimo identycznej liczby wierzchołków, analogiczna sytuacja miała miejsce w przypadku plików gr17.txt i tsp_17.txt (Rysunek 5). Prawdopodobnie wynikało to z faktu, że wartości w macierzy sąsiedztwa dla pliku burma14.txt i gr17.txt różnią się między sobą wartością bardziej niż dla pliku tsp_14.txt i tsp_17.txt. Dla tsp_14.txt i tsp_17.txt przedział ten jest rzędu dziesiątek, a dla burma14.txt i gr_17.txt rzędu setek. Z uwagi na wspomniany powyżej fakt, oraz to, że dla pliku gr_21.txt nie zakończyły się obliczenia w czasie przekraczającym 3 godziny oraz pomijalnie niski dla tsp_10.txt, zdecydowano aby wybrać zestaw instancji ze strony http://jaroslaw.mierzwa.staff.iiar.pwr.wroc.pl/pea-stud/tsp/ wyłączając tsp_6_1.txt i tsp_6_2.txt. Liczby w macierzach instancji z wyżej wspomnianej strony różniły się między sobą w mniejszym stopniu niż w plikach burma14.txt oraz gr17.txt (przedziały były maksymalnie rzędu dziesiątek) oraz dostępne były tam także instancje już od 6 wierzchołków, a nie dopiero od 14. Wybrano zatem 6 instancji, które podczas wstępnych testów nie dostarczyły powodów do przypuszczania, że któraś z nich zawiera cechę mogącą sprawić, że pomiary będą niemiarodajne. Co istotne, wzrost czasu między burma_14.txt oraz gr_17.txt pozwala przypuszczać, że gdyby były dostępne mniejsze instancje o podobnym przedziale wartości krawędzi to ich linia trendu była by podobna jak dla zaprezentowanych w punkcie 6 wyników. Z uwagi na fakt, że dwie instancje to za mało, aby pomiary były rzetelne, zdecydowano o wybraniu do pomiarów wyżej wspomnianego zestawu instancji (plik inicjujący pomiary.ini).



Rysunek 4: Porównanie czasu wykonania tsp_14.txt i burma14.txt



Rysunek 5: Porównanie czasu wykonania tsp_17.txt i gr17.txt

Do przeprowadzenia pomiarów wybrano następujący zestaw instancji:

-tsp_10.txt, min. koszt: 212

-tsp_12.txt, min. koszt: 264

-tsp_13.txt, min. koszt: 269

-tsp_14.txt min. koszt: 282

-tsp_15.txt min. koszt: 291

-tsp_17.txt min. koszt: 39

http://jaroslaw.mierzwa.staff.iiar.pwr.wroc.pl/pea-stud/tsp/

5.Procedura badawcza

Należało zbadać zależność czasu rozwiązania problemu oraz zużycie pamięci od wielkości instancji. W przypadku algorytmu realizującego algorytm breach and bound dla przestrzeni rozwiązań dopuszczalnych nie występowały parametry programu, które mogły mieć wpływ na czas i jakość uzyskanego wyniku. W związku z tym procedura badawcza polegała na uruchomieniu programu sterowanego plikiem inicjującym .ini: (format pliku: liczba_wykonań, optymalny_koszt_cyklu_Hamiltona, nazwa_pliku_z_grafem, [ścieżka optymalna]). Jeśli nie było podanej ścieżki optymalnej, wpisane było 0, które program interpretuje jako informację, aby nie pokazywać komunikatu o niezgodności cyklu Hamiltona z oczekiwanym. Miało to miejsce tylko w przypadku wstępnych testów sprawdzających poprawność działania i orientacyjne czasy wykonania różnych instancji, dla właściwych pomiarów istniały ścieżki dla każdej instancji. Wyniki były zapisywane w pliku pomiary.csv. Poniżej przedstawiono zawartość pliku inicjującego:

50 212 tsp_10.txt 0 3 4 2 8 7 6 9 1 5 0

50 264 tsp_12.txt 0 1 8 4 6 2 11 9 7 5 3 10 0

50 269 tsp_13.txt 0 10 3 5 7 9 11 2 6 4 8 1 12 0

50 282 tsp_14.txt 0 10 3 5 7 9 13 11 2 6 4 8 1 12 0

50 291 tsp_15.txt 0 12 1 14 8 4 6 2 11 13 9 7 5 3 10 0

50 39 tsp_17.txt 0 11 13 2 9 10 1 12 15 14 5 6 3 4 7 8 16 0

Każda instancji rozwiązywana była zgodnie z liczbą jej wykonań, w tym przypadku każda 50 razy. Do pliku wyjściowego pomiary.csv zapisywany był czas wykonania pomiarów oraz nazwa instancji której te pomiary dotyczą. W przypadku możliwej niezgodności kosztu lub ścieżki z optymalną, program wyświetlał na ekranie stosowny komunikat. W przypadku przeprowadzonych pomiarów wszystkie koszty i ścieżki były właściwe. Plik wyjściowy zapisywany był w formacie csv. Dane znajdują się w następującym formacie nazwa_instancji; czas_1_pomiaru; czas_2_pomiaru; … ; czas_n_pomiaru. Program dla instancji (0,14> oblicza czas w milisekundach, dla instancji (14, +∞) w sekundach. Program po zakończeniu obliczeń wyświetla informację na ekranie informującą o zastosowanych w zapisie jednostkach dla podanych wyżej przedziałów instancji. Poniżej przedstawiono fragment zawartości pliku wyjściowego (jednostki w nawiasach kwadratowych zostały dopisane wtórnie, w celach informacyjnych):

tsp_12.txt;171;171;171 [milisekundy]

tsp_15.txt;12;12;12;13;13 [sekundy]

Pomiary zużycia pamięci zostały przeprowadzone przy użyciu menedżera zadań systemu Windows 10. Dla każdej instancji pomiar był przeprowadzany poprzez wczytanie pliku inicjującego nakazującego programowi wykonanie algorytmu breach and bound dla tej instancji 10000 razy. Poprawność ścieżek nie była sprawdzana, gdyż miało to miejsce już wcześniej i nie było takiej konieczności. Stąd w miejscu ścieżki liczba 0 informująca program, aby pomijał ten parametr z pliku .ini. Zadeklarowanie dużej liczby powtórzeń miało na celu zapewnienie czasu na wykonanie ręcznego odczytu używanej przez proces pamięci RAM. Poniżej przykład zawartości pliku inicjującego dla tsp_10.txt do pomiarów zużycia pamięci:

10000 212 tsp_10.txt 0

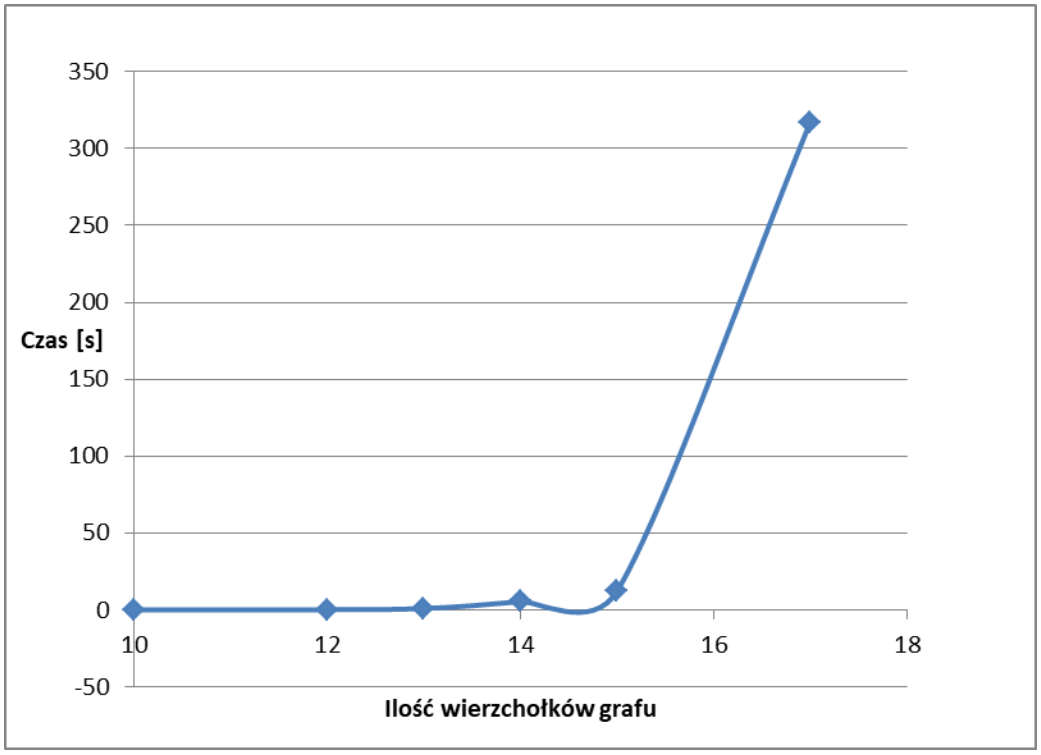
Wyniki zostały opracowane w MS Excel.

6.Wyniki

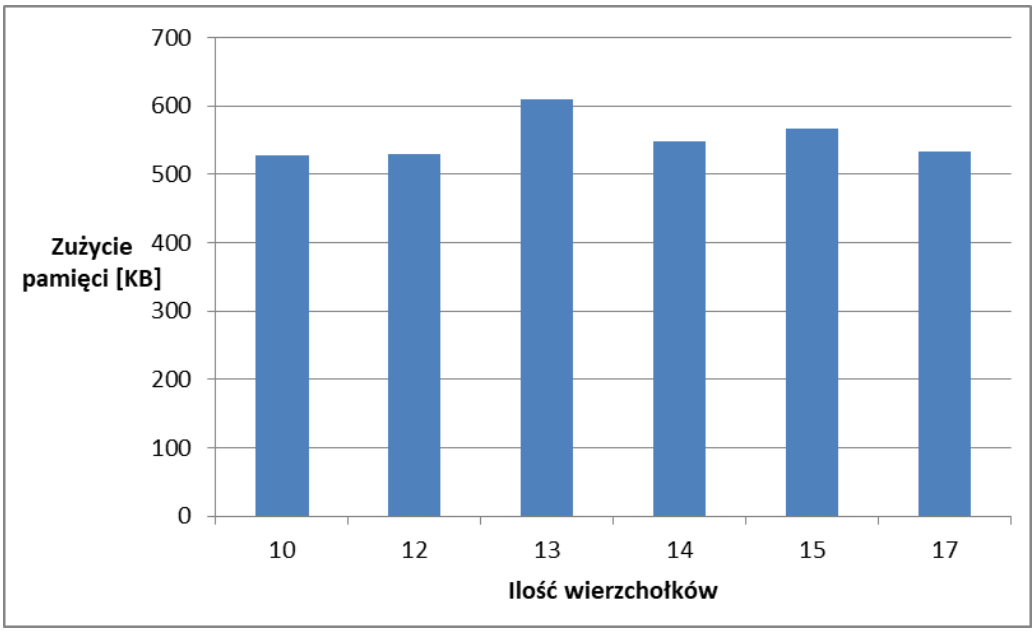
Wyniki zgromadzone zostały w plikach: pomiary_czasu.csv, testy_wstepne.csv, pomiary_pamieci.xlsx Wszystkie ww. pliku zostały dołączone do raportu i znajdują się na dysku Google pod adresem:

https://drive.google.com/drive/folders/113X275AvvonhyTK1MFhEiYMs25LdP0iI?usp=sharing

Wyniki dla algorytmu breach and bound przedstawione zostały w postaci wykresu zależności czasu uzyskania rozwiązania problemu od wielkości instancji (Rysunek 6) oraz w postaci wykresu zależności zużycia pamięci RAM przez program od wielkości instancji (Rysunek 7).



Rysunek 6: Zależność wielkości instancji od czasu wykonania dla algorytmu breach and bound



Rysunek 7: Zużycie pamięci dla badanych instancji dla algorytmu breach and bound

Pomiary zostały przeprowadzone na sprzęcie o następujących parametrach:

- Procesor Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz 2.50 GHz
- RAM 8GB
- System operacyjny Windows 10 64-bit, procesor x64

Czas został zmierzony za pomocą biblioteki std::chrono, natomiast zużycie pamięci za pomocą menedżera zadań systemu Windows 10.

7.Analiza wyników i wnioski dot. algorytmu breach and bound

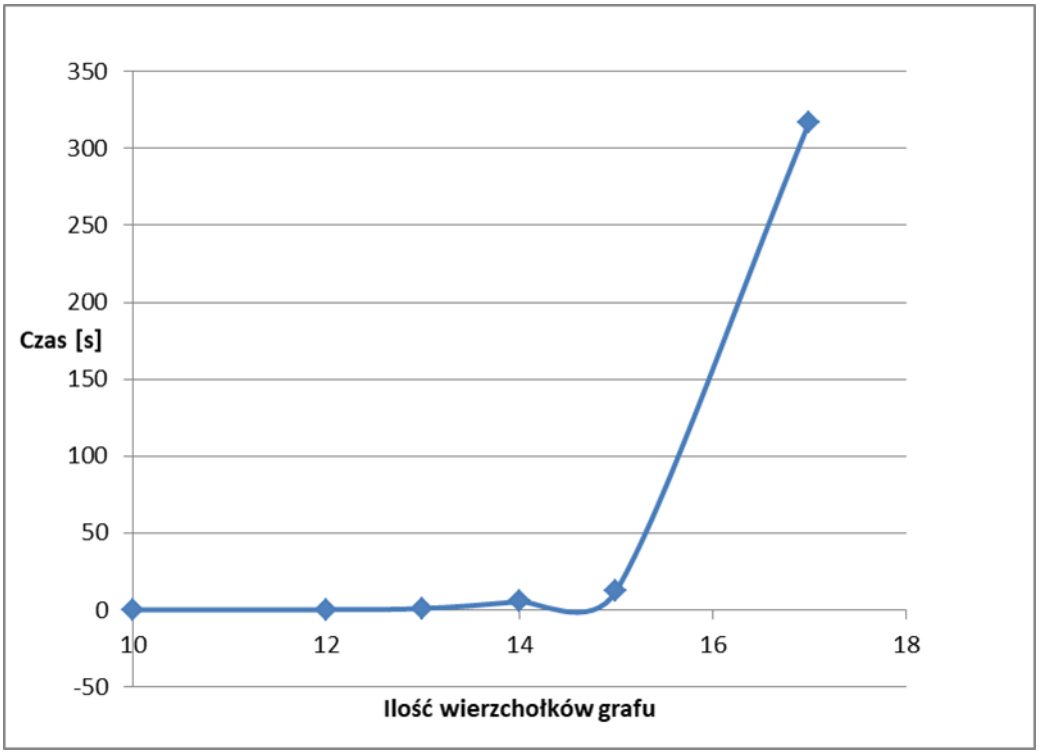
Złożoność czasowa algorytmu breach and bound dla badanych instancji jest zgodna z oczekiwaną i jest wyraźnie mniejsza od O(n!) oraz większa od O(1). Potwierdza to rysunek 6 pokazujący zależność ilości wierzchołków od czasu dla algorytmu breach and bound. Zaimplementowany przez mnie algorytm początkowy koszt najmniej kosztownego znanego cyklu szacuje za pomocą heurystycznego algorytmu najbliższego sąsiada. Dzięki temu nie ma konieczności ustawiania tej wartości na INT_MAX i jest to pierwszy czynnik, który ogranicza ilość koniecznych operacji i poprawia przez to wydajność. Następnie algorytm porównuje z każdym następnym dodanym wierzchołkiem dotychczasowy koszt aktualnie rozważanego potencjalnego cyklu Hamiltona z najmniejszym znanym kosztem (zmienna bestCost) i jeśli jest on większy od bestCost przed zakończeniem sprawdzania całości to nie sprawdza reszty potencjalnego cyklu, czyli obcina liść. Jest to drugi czynnik, który ogranicza ilość operacji i niezbędny do ich wykonania czas. W miarę spadku najmniejszego znanego kosztu, sprawdzanych do końca jest też coraz mniej potencjalnych cykli i jest to następny czynnik, który zmniejsza ilość potrzebnego do wykonania algorytmu czasu. Gdyby algorytm sprawdzał wszystkie kombinacje, jego złożoność była by klasy O(n!). Jednak z uwagi na używane przez metodę breach and bound mechanizmy zawsze będzie ona zawsze nie większa od O(n!), a zazwyczaj znacznie mniejsza. Złożoność dla danej instancji zależy od tego w jakim stopniu i jak wiele permutacji mogących być cyklem Hamiltona jest sprawdzane. Im mniej permutacji jest sprawdzanych i w im mniejszym stopniu, tym złożoność będzie mniejsza.

Warto również zwrócić uwagę na wpływ szerokości przedziału w jakim znajdują się wagi krawędzi grafu na czas wykonania. Podczas wstępnego testowania algorytmu zaobserwowano zależność, że im większy jest ten przedział, tym dłuższy jest czas wykonania algorytmu breach and bound. Dla dwóch par instancji o takiej samej ilości wierzchołków ale różnej szerokości przedziału liczb w jakiej znajdują się wagi ich krawędzi otrzymano istotnie różniące się wyniki, które spowodowały decyzję, o wybraniu zestawu którego użyto do pomiarów i uwzględnienia w tej decyzji tego czynnika. Na rysunku 6 można zaobserwować, że z poddanych badaniu instancji wszystkie wykazują ten sam trend, z czego płynie wniosek, że dobrane do pomiarów instancje były reprezentatywne pod względem przedziału w jakim znajdują się wagi krawędzi grafów.

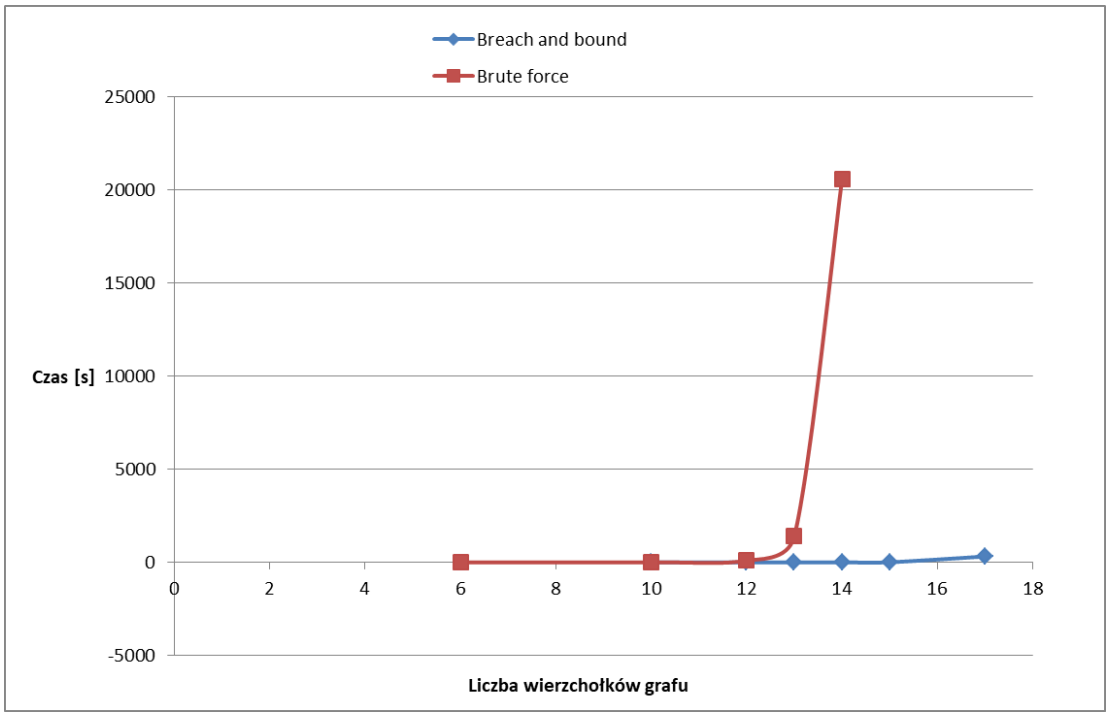
Obserwowane na rysunku 7 zużycie pamięci RAM w trakcie działania programu dla badanych instancji było nieskorelowane z wielkością instancji i pozostawało niezmiennie na podobnym, niskim poziomie. Można przypuszczać, że wynika to z faktu, że algorytm nie przechowuje dużych ilości danych, mój algorytm przechowuje jedynie informacje o grafach i wynikach prowadzonych obliczeń. Co więcej, szeroko używana przeze mnie do kluczowych operacji zmienna vector jest zoptymalizowana pod kątem używania pamięci, co w połączeniu z charakterem algorytmu B&B który nie ma tendencji do używania dużych ilości pamięci wyjaśnia otrzymane wyniki pomiarów.

8.Porównanie algorytmów breach and bound i brute force

W porównaniu do algorytmu brute force, algorytm breach and bound jest wielokrotnie szybszy (Rysunek 9). Powodem tego stanu rzeczy jest fakt, że algorytm brute force sprawdza dokładnie wszystkie permutacje wierzchołków danego grafu i w ten sposób wyszukuje najmniej kosztowny cykl Halmiltona (w przypadku problemu minimalizacyjnego, który był przedmiotem badań dla obu algorytmów z Rysunku 9). Algorytm breach and bound stosuje zaś ograniczenia (bound) i obcinanie liści (breach). Dzięki temu nie musi sprawdzać dokładnie wszystkich permutacji, a co za tym idzie znacznie ograniczana jest ilość wykonywanych operacji, a więc także zużywanego w tym celu czasu.



Rysunek 8: Zależność wielkości instancji od czasu wykonania dla algorytmu breach and bound



Rysunek 9: Porównanie czasu wykonania w zależności od ilości wierzchołków dla algorytmu brute force i breach and bound