Programowanie Efektywnych Algorytmów - Projekt: Sprawozdanie

Wydział Elektroniki	**Kierunek: Informatyka Techniczna**
Grupa zajęciowa Wt 17:05	Semestr: 2021/2022 Zima
Prowadzący:	Dr inż. Antoni Sterna

Autor
Byczko Maciej 252747

Wstęp teoretyczny

Problem komiwojażera

Problem komiwojażera (ang. travelling salesman problem, TSP) – zagadnienie optymalizacyjne, polegające na znalezieniu drogi o najmniejszym koszcie.

komiwojażer - przedstawiciel firmy podróżujący w celu zdobywania klientów i przyjmowania zamówień na towar. (definicja ze słownika)

W celu zobrazowania problemu należy wyobrazić sobie tytułowego komiwojażera, który podróżuje między miastami w celu wykonywania swojej pracy. Podróż zaczyna z siedziby swojej firmy po czym jego trasa przebiega przez każde miasto dokładnie jeden raz, aż w końcu wraca z powrotem do głównego budynku firmy.

Matematycznie prezentujemy ten problem jako graf którego wierzchołki są miastami a łączące je trasy to krawędzie z odpowiednimi wagami. Jest to pełny graf ważony oraz może być skierowany, co tworzy problem asymetryczny.

Rozwiązanie problemu komiwojażera sprowadza się do znalezienia właściwego - o najmniejszej sumie wag krawędzi - cyklu Hamiltona, czyli cyklu przechodzącego przez każdy wierzchołek grafu dokładnie jeden raz. Przeszukanie wszystkich cykli (czyli zastosowanie metody *Brute Force*(przegląd zupełny)) nie jest optymalną metodą, jako że prowadzi do wykładniczej złożoności obliczeniowej - $O(n!)$, dla której problemy o dużym n traktowane jako nierozwiązywalne. Klasyfikuje to problem komiwojażera jako problem NP-trudny, czyli niedający rozwiązania w czasie wielomianowym. To powoduje konieczność skorzystania z tzw. algorytmów heurystycznych bądź metaheurystycznych (bardziej ogólnych), a w naszym przypadku konkretnie algorytmu *Branch & Bound* oraz *programowania dynamicznego*.

Przegląd zupełny (Brute Force)

Przegląd zupełny generuje i sprawdza wszystkie możliwe kombinacje i wybiera pierwszą z nich (gdyż może być kilka rozwiązań), która spełnia warunek posiadania najmniejszej wartości. Ponieważ sposób ten sprawdza każde możliwe rozwiązanie to mamy gwarancję znalezienia poprawnego rozwiązania lecz ogromną wadą jest złożoność obliczeniowa $O((n-1)!)$, która sprawia że dla dużych wartości n rozwiązanie problemu tą metodą jest nieoptymalne gdyż wymaga to od nas niewyobrażalnych ilości czasu do sprawdzenia każdej możliwej permutacji.

Programowanie dynamiczne (DP - Dynamic Programming)

Programowanie dynamiczne jest algorytmem przyśpieszonym, który jest algorytmem dokładnym o złożoności obliczeniowej $O(n^2 2^n)$. Koncepcja algorytmu opiera się na upraszczaniu skomplikowanych problemów do momentu aż staną się dla nas trywialne, które w przypadku TSP będą bezpośrednim pobraniem wartości z macierzy reprezentatywnej.

Metoda podziału i ograniczeń (B&B - Branch and Bound)

Algorytm Branch & Bound, czyli metoda podziału i ograniczeń opiera się na przeszukiwaniu drzewa reprezentującego przestrzeń rozwiązań problemu. Dzięki tak zwanym „odcięciom” można znacznie zredukować liczbę przeszukiwanych wierzchołków. Nazwa Branch & Bound wiele mówi o ogólnej koncepcji algorytmu - rozgałęzianie (ang. branching) tworzy następników (synów) danego wierzchołka, a ograniczanie (ang. bounding) odcina - czyli ściślej rzecz biorąc pomija - gałęzie, które nie doprowadzą nas do optymalnego rozwiązania.

Poszczególne odmiany Branch & Bound mogą różnić się niemalże wszystkimi parametrami - poprzez różne strategie przeszukiwania, bardziej i mniej efektywne funkcje liczenia dolnego ograniczenia (ang. lower bound) oraz różne mechanizmy dające nam szybciej górne ograniczenie (ang. upper bound).

Przykłady praktyczne

BruteForce

W implementacji wykorzystaliśmy tzw. algorytm `Next Permutation` polegający na generowaniu kolejnych permutacji, metoda ta została napisana na bazie `next_permutation` z biblioteki STL z języka C++.

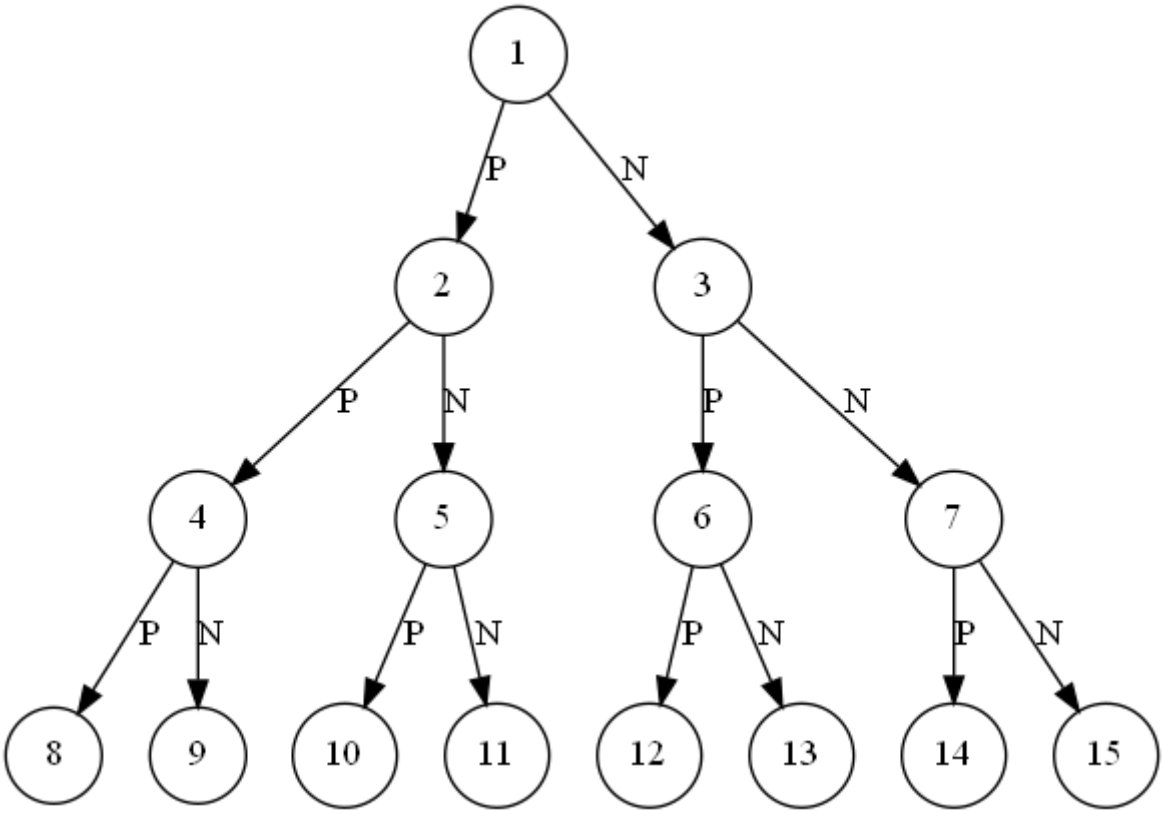
Polega ona na generowaniu permutacji rekursywnie:

```
perm(ab) ->

a + perm(b) -> ab
b + perm(a) -> ba

Dla każdego znaku zwróć ten znak oraz kombinację pozostałych znaków:

perm(abc) ->

a + perm(bc) -> abc, acb
b + perm(ac) -> bac, bca
c + perm(ab) -> cab, cba

etc.
```

Następnie wyliczaliśmy koszt każdej kombinacji i na koniec zwróciliśmy najlepszą z nich.

Metoda podziału i ograniczeń

`Branch & Bound` opiera się na przeszukiwaniu drzewa reprezentującego przestrzeń rozwiązań problemu. Dzięki tak zwanym „odcięciom” można znacznie zredukować liczbę przeszukiwanych wierzchołków. Nazwa `Branch & Bound` wiele mówi o ogólnej koncepcji algorytmu - rozgałęzianie (ang. branching) tworzy następników (synów) danego wierzchołka, a ograniczanie (ang. bounding) odcina - czyli ściślej rzecz biorąc pomija - gałęzie, które nie doprowadzą nas do optymalnego rozwiązania.



Na przykładowym drzewie można opisać zastosowane metody:

Przeszukiwanie wszerz

Gdy zaczynamy w korzeniu (1) to do naszej kolejki priorytetowej dodajemy sąsiadów (2,3) po sprawdzeniu czy spełniają one warunki lower oraz upper bound. Następnie bierzemy następną wartość z kolejki (2) i także sprawdzamy są sąsiadów (4,5) czy spełniają warunki, następnie dodajemy ich do kolejki.

Wykonujemy to tak długo dopóki kolejka się nie wyczerpie.

Przeszukiwanie wgłąb

Gdy zaczynamy w korzeniu (1) to do stosu dodajemy sąsiadów pierwszego znalezionego sąsiada (2) po sprawdzeniu czy spełnia on warunki lower oraz upper bound.

Następnie bierzemy nowo znalezionego sąsiada i dodajemy jego pierwszego znalezionego sąsiada (4) po sprawdzeniu ograniczeń.

Wykonujemy to tak długo dopóki kolejka się nie wyczerpie.

Wyznaczenie ograniczeń

Upper bound wyznacza się zwykle za pomocą metody "Najbliższego sąsiada" czyli od korzenia wybieramy najniższą wartość, następnie przechodzimy do znalezionego wierzchołka i u niego sprawdzamy najmniejszą wartość.

Wykonujemy to tak długo aż nie wyczerpią nam się wierzchołki. Na koniec nie wolno zapomnieć o dodaniu przejścia do 0 oraz +1 do wartości aby nie została pominięta gdyby okazało się że to jest optymalne rozwiązanie.

Lower bound jest bardziej skomplikowane, najczęściej wylicza się koszt wejścia i wyjścia z wierzchołka na podstawie najniższych wag od niego wychodzących, źle zaimplementowany może spowolnić algorytm zamiast go przyśpieszyć gdyż wyliczenie będzie trwało więcej niż sprawdzenie ścieżki.

Programowanie Dynamiczne

Koncepcja programowania dynamicznego opiera się na dzieleniu skomplikowanego problemu na mniejsze pod-problemy, aż do momentu gdy nasze pod-problemy stają się trywialne. Każdy rozwiązany pod-problem jest zapamiętywany, dzięki czemu nie trzeba rozwiązywać go wielokrotnie, wystarczy wziąć wcześniej otrzymany wynik. Prostym przykładem jest obliczenie n-tej liczby ciągu Fibonacciego. Wzór rekurencyjny:

$$fib(n) = fib(n-1) + fib(n-2)$$

Licząc np. $fib(5)$, musimy obliczyć $fib(4)$ i $fib(3)$, żeby mieć $fib(4)$, musimy obliczyć $fib(3)$ i $fib(2)$ etc. Już tutaj widać, że liczenie np. $fib(3)$ odbędzie się więcej niż jeden raz, oraz łatwo zauważyć że dla bardzo dużego n bardzo dużo działań będzie się powtarzać. Za pomocą programowania dynamicznego, możemy zapamiętać każde obliczone $fib(n-i)$, i zamiast obliczać funkcję ponownie, wziąć wcześniej obliczoną wartość.

Aby rozwiązać problem komiwojażera tym sposobem, musimy znaleźć sposób dzielenia problemu na pod-problemy. Dla przykładu, mając 4 miasta, komiwojażer musi zacząć w mieście 1, przejść przez miasta 2, 3, 4 i wrócić do 1 najkrótszą ścieżką. Teraz mamy pod-problem, musimy znaleźć minimum z trzech opcji:

droga `1->2` + najkrótsza ścieżka od 2, prowadząca przez 3, 4, kończąca na 1

droga `1->3` + najkrótsza ścieżka od 3, prowadząca przez 2, 4, kończąca na 1

droga `1->4` + najkrótsza ścieżka od 4, prowadząca przez 2, 3, kończąca na 1

Rozwijając np. `1->2`, musimy znaleźć kolejne minimum z opcji:

`1->2->3` + najkrótsza ścieżka od 3, prowadząca przez 4, kończąca na 1

`1->2->4` + najkrótsza ścieżka od 4, prowadząca przez 3, kończąca na 1

Rozwijając `1->2->3`, mamy:

`1->2->3->4` + najkrótsza ścieżka od 4, prowadząca przez $\varnothing$, kończąca na 1

Ścieżka 4->1 jest nam znana, więc jest to problem trywialny. Wracając do pod-problemu wyżej (1->2->3 + minimum), mamy 3->4->1, 4->1 znamy z problemu niżej, dodajemy do drogi 3->4 i mamy kolejny rozwiązany pod-problem. Musimy rozwiązać również problem 1->2->4 + minimum, wybrać z nich obu minimum, i to „przekazać" do problemu wyżej, etc.

Zatem ogólny wzór można określić tak:

$$f(i, S) = min(d_i \to k + f(k, S-k))$$

gdzie i to miasto startowe, S to zbiór miast, przez które trzeba przejść, k to dowolne miasto ze zbioru S, a $d_i \to k$ to droga pomiędzy miastami i oraz k.

Programowanie dynamiczne wykorzystujemy do zapamiętywania obliczonych już rozwiązań, za pomocą tablicy dwuwymiarowej o rozmiarach $2n-1$ (-1, ponieważ pierwsze miasto jest już ustalone) na n. Zbiory

będą reprezentowane jako maski bitowe, tzn. kolumna np. o numerze 10 reprezentuje zbiór zawierający miasta 3 i 1 (licząc od 0), ponieważ $10_{10} = 1010_2 = 2^3 + 2^1$. Wiersze reprezentują początkowe wierzchołki, tzn. wiersz 2 i kolumna 10 trzyma w sobie wagę trasy zaczynającej się od miasta 2, przechodzącej przez zbiór {1, 3} i kończącej na mieście 0.

Opis implementacji algorytmów

Plan eksperymentu

Program został napisany w języku `C#`, w `.NET` Framework, w środowisku `JetBrains Rider`.

Ponieważ .NET nie zawiera w swoich kolekcjach kolejki priorytetowej to w programie została użyta kolekcja z pakietu MedallionPriorityQueue

Badane rozmiary problemu to od 3 do 12 dla wszystkich algorytmów oraz większe rozmiary dla tych rozwiązań w których czasy były optymalne.

Wagi krawędzi były losowane z przedziału `[1,100]`.

Do mierzenia czasu wykorzystano klasę `StopWatch` z przestrzeni nazw `System.Diagnostics` mierzącą czas z dokładnością do nanosekund lecz my przeliczyliśmy to na milisekundy.

Test każdego algorytmu został wykonany 100 razy.

Do każdego testu generowano nowy, losowy graf.

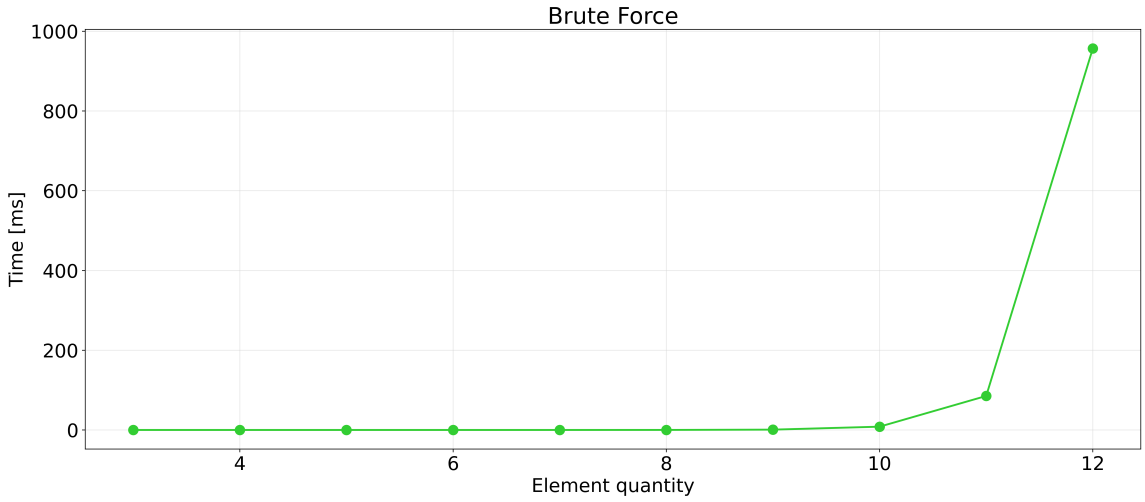
wyniki eksperymentu

Czasy podane w milisekundach.

Liczba wierzchołków	BruteForce	B&B Breath	B&B Deep	DynamicProgramming
3	0.008803	0.054176	0.014542	0.009216
4	0.000879	0.003812	0.003178	0.001199
5	0.002427	0.010186	0.0073	0.002268
6	0.003544	0.028519	0.019124	0.005625
7	0.015227	0.114645	0.057516	0.014581
8	0.121267	0.347821	0.188279	0.042642
9	0.841295	1.212657	0.798896	0.066363
10	8.283555	4.732777	2.472275	0.160255
11	85.317322	13.926271	9.531186	0.425354
12	956.838701	49.033874	27.860567	0.910719
13	-	174.638263	77.712093	2.465029
14	-	553.21148	263.692528	5.161136

Liczba wierzchołków	BruteForce	B&B Breath	B&B Deep	DynamicProgramming
15	-	1780.36111	779.636625	11.546628
16	-	6253.766069	2552.038531	27.92583
17	-	-	-	63.373945
18	-	-	-	148.910532
19	-	-	-	350.543253
20	-	-	-	912.469592

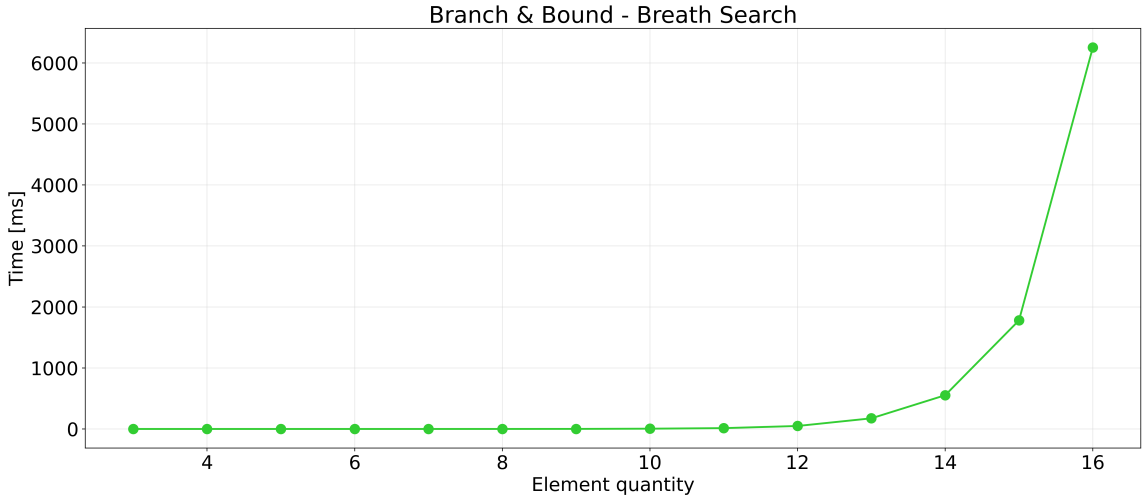
Wyniki BruteForce

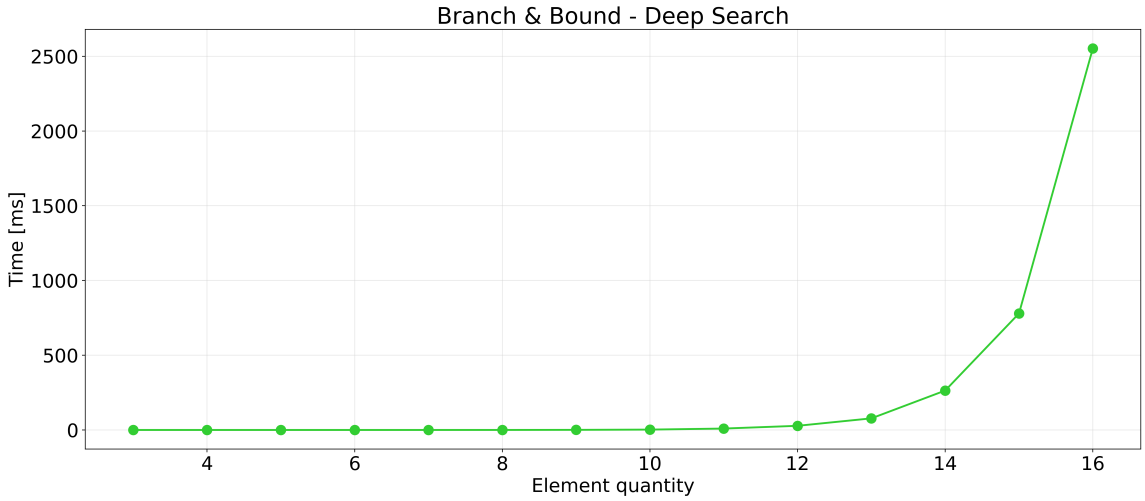


Złożoność przeglądu zupełnego okazuje się być bliska prawdzie gdyż proporcja czasowa wychodzi taka jakiej oczekiwaliśmy (np. 956/85 ~= 11 dla problemu o wielkości 12).

Wyniki Branch And Bound

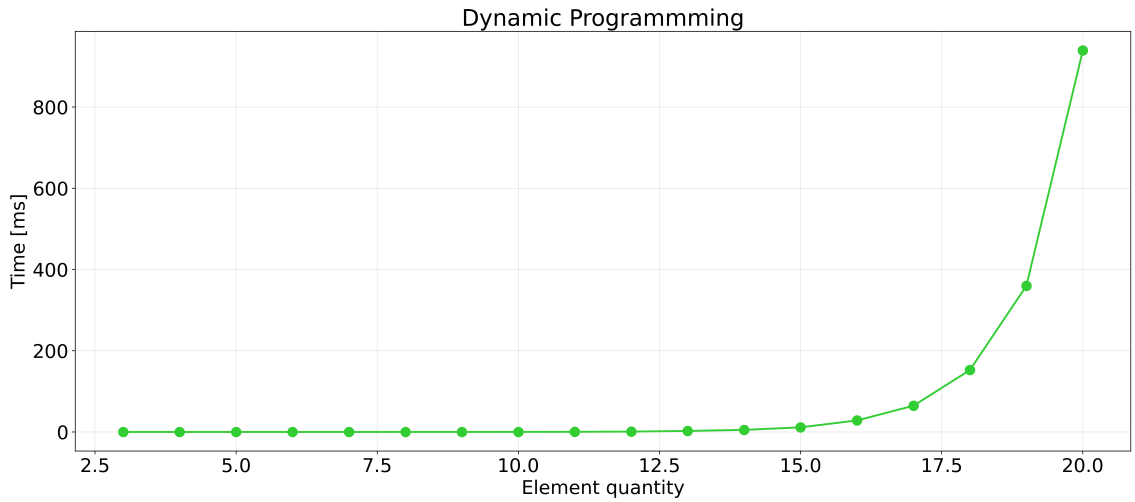
Ze względu na specyfikę algorytmu - m.in. liczne odcięcia - nie sposób ustalić złożoności obliczeniowej go reprezentującej.





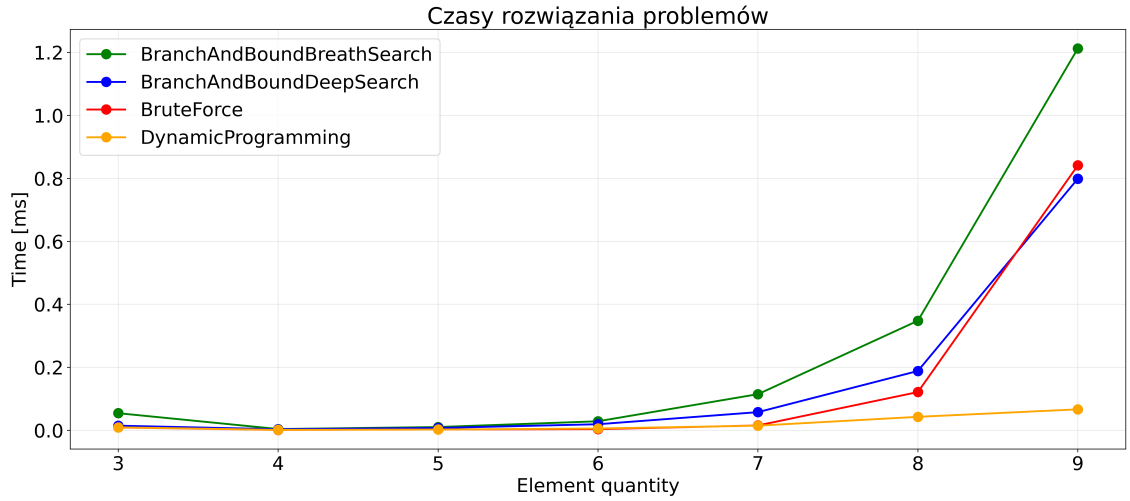
Dla Branch & Bound można zauważyć że w danej implementacji lepiej wypada Depth first.

Wyniki Dynamic Programming

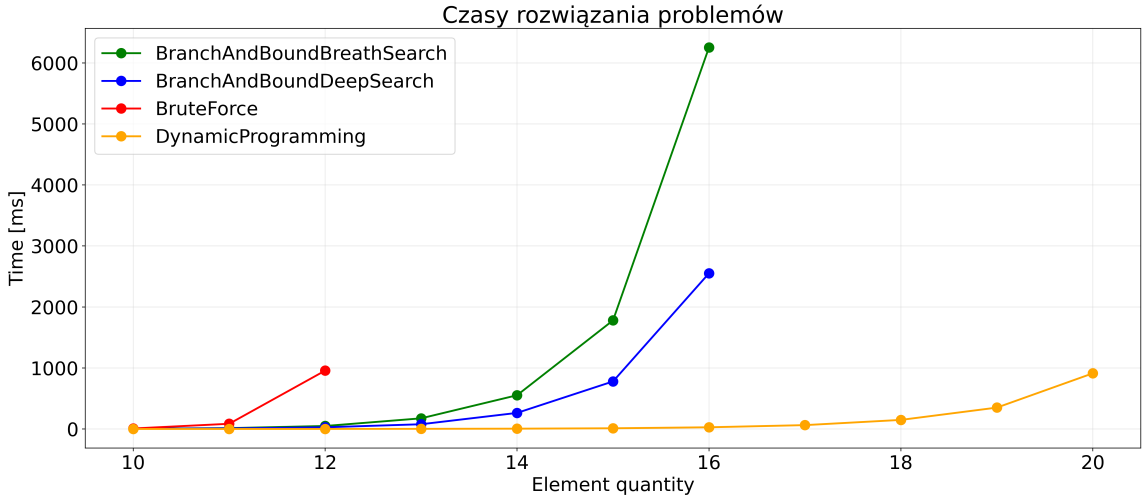


Programowanie dynamiczne także zgadzało się z teorią gdyż na podstawie złożoności $n^2 2^n$ można sprawdzić że $\frac{n^2 2^n}{(n-1)^2\ 2^(n-1)} = 2\frac{n^2}{(n-1)^2}$ wartości powinny rosnąć lekko ponad dwukrotnie ($2\frac{17^2}{(17-1)^2} = 2.25$ a proporcja czasów wynosi $63/27=2.33$)

Porównanie algorytmów dla małych instancji



Przegląd zupełny dla najmniejszych badanych rozmiarów problemu okazywał się najszybszym algorytmem. Dla takich rozmiarów możliwych permutacji jest wystarczająco mało, że opłacało się po prostu sprawdzić wszystkie. Jednak już dla rozmiaru powyżej 7, Brute Force był wolniejszy od pozostałych algorytmów.



Przegląd zupełny dla rozmiaru powyżej 12 wykonywał się zbyt długo ,aby móc przeprowadzić test dla takiego rozmiaru 100 razy w rozsądnym czasie.

Wnioski

Z dwóch porównywanych metod podziału i ograniczeń, szybszą okazała się metoda przeszukiwania w głąb.

Najszybszym algorytmem okazało się programowanie dynamiczne. Jednakże trzeba pamiętać, że odbywa się to kosztem pamięci - złożoność pamięciowa rośnie wykładniczo.

Bibliografia

http://antoni.sterna.staff.iiar.pwr.wroc.pl/pea/PEA_wprowadzenie.pdf

https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/

https://stackoverflow.com/questions/756055/listing-all-permutations-of-a-string-integer

https://www.ii.uni.wroc.pl/~prz/2011lato/ah/opracowania/met_podz_ogr.opr.pdf

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