Projektowanie Efektywnych Algorytmów

Kierunek *Informatyka*	**Termin** *Poniedziałek 15:15*
Temat *Metoda podziału i ograniczen i/lub programowanie dynamiczne.*	**Problem** *TSP*
Skład grupy *Kinga Marek 235 280*	**Nr grupy** -
Prowadzący *Mgr inż. Radosław Idzikowski*	**data** *December 6, 2019*

1 Opis problemu

Problem komiwojażera (ang. *The travelling Salesman Problem*) to problem optymalizacyjny polegający na znalezieniu w grafie cyklu, który zawiera wszystkie jego wierzchołki (dokładnie jeden raz), a suma wag tego cyklu jest najmniejsza. W bardziej formalnym ujęciu problem ten polega na znalezieniu minimalnego cyklu Hamiltona w grafie.
Nazwa problemu wywodzi się z typowej ilustracji problemu, która przedstawia komiwojażera podróżującego pomiędzy określoną ilością miast w których sprzedaje swoje produkty lub zawiera różne oferty handlowe. Na koniec komiwojażer powraca do swojego miasta rodzinnego. Komiwojażer chce, aby jego trasa była możliwe jak najkrótsza - zagadnienie to rozwiązuje właśnie problem komiwojażera. W rzeczywistości jednak problem ten ma bardzo szerokie zastosowanie w optymalizacji wielu procesów.

2 Metody rozwiązania

2.1 O zastosowanej technologii i architekturze (dodatkowo)

O języku, jakim jest Go

Do implementacji projektu został wykorzystany język Go. Jest to język programowania opracowany przez Google, a składnią wzorowany jest na języku C. Charakteryzuje się prostotą oraz bardzo dużym wsparciem dla programowania współbieżnego i sieciowego. Język ten posiada bardzo szybki kompilator na platformę x86, x64 i ARM oraz garbage collector zwalniający z narzutu jakim jest dbałość o zwalnianie pamięci. Go nie jest typowym, obiektowym językiem programowania. Nie posiada klas, lecz struktury i nie umożliwia dziedziczenia. W miejsce dziedziczenia proponowana jest kompozycja. Polimorfizm możliwy jest tylko poprzez implementowanie określonego interfejsu. Hermetyzacja możliwa jest tylko w obrębie pakietu (ang. *package*).

O stworzonej architekturze

Jako osoba pracująca na co dzień w obiektowym języku jakim jest C# zaplanowanie architektury okazało się zadaniem trudniejszym niż można się było tego początkowo spodziewać. Zdecydowano się na zaprojektowanie interfejsu, który implementuje każdy metodę rozwiązującą problem komiwojażera wybranym algorytmem.Algorytm ten posiada tylko jedną metodę, która zwraca najlepszą znalezioną ścieżkę.

```
// TspAlgorithm defines a way to solve travelling salesman problem
type TspAlgorithm interface {
        Resolve(adjacencyMatrix [][]int) []int
}
```

Interfejs ten jest elementem klasy $TravellingSalesmanProblem$, która zawiera wszystkie dane potrzebne do scharakteryzowania instancji problemu, rozwiązania używając algorytmu zaimplementowanego przez interfejs *TspAlgorithm*, a także metody potrzebne do wczytania grafu z pliku.

```
// TravellingSalesmanProblem defines a complete data
// and an algorithm used to solve the problem.
type TravellingSalesmanProblem struct {
        Name            string
        Size            int
        AdjacencyMatrix [][]int
        Solution        []int
        MinimumCost     int
        CalculationTime time.Duration
```

```
        Algorithm       TspAlgorithm
}
```

Dla implementowanych algorytmów tworzono oddzielne struktury, które implementowały interfejs *TspAlgorithm*. Dla większej czytelności struktury te wyposażone były w inne metody pomocnicze. Dzięki temu możliwe było wygodne mierzenie czasu rozwiązania określonej instancji problemu dla rożnych algorytmów - wystarczyła, zmiana implementacji interfejsu *TspAlgorithm*. W pewien sposób może to przypominać implementację wzorca projektowego strategia w Go.

2.2 Metoda zachłanna - Brute Force

Algorytm Brute Force dokonuje przeglądu zupełnego tworząc każdą możliwą ścieżkę i sprawdzając, czy otrzymany cykl ma koszt przejścia mniejszy od najlepszego znalezionego dotychczas. Wykonanie algorytmu rozpoczanamy od pierwszego wywołania rekurencyjnej funkcji przeszukującej przestrzeń stanów. Jako argument *path* przechowujący dotychczasowo zbudowaną ścieżkę przekazujemy pustą tablicę (w Go jest to formalnie slice) oraz listę wszystkich wierzchołków grafu (jeszcze żaden nie został odwiedzony).

```
// FindBestPathRecursively is a recursive function
// that finds all cycles in graph using search tree
func (b *BruteForce) FindBestPathRecursively(path []int, notVisitedNodes []int) {

        if len(notVisitedNodes) > 0 {
                for index, node := range notVisitedNodes {
                        SwapLastAndIndex(notVisitedNodes, index)
                        b.FindBestPathRecursively
                            (append(path, node), notVisitedNodes[:len(notVisitedNodes)-1])
                        SwapLastAndIndex(notVisitedNodes, index)
                }
        } else {
                cost := b.TargetFunction(path)
                if cost < b.minimumCost {
                        b.bestPath = make([]int, len(path))
                        copy(b.bestPath, path)
                        b.minimumCost = cost
                }
        }
}
```

W każdym rekurencyjnym wywołaniu sprawdzamy, czy zostały odwiedzone już wszystkie wierzchołki (czy lista *notVisitedNodes* jest pusta). Jeśli istnieją jeszcze nieodwiedzone wierzchołki to dla każdego nieodwiedzonego wierzchołka tworzymy nowe rekurencyjne wywołanie z wybranym wierzchołkiem dodanym jako kolejny element ścieżki. W celach optymalizacyjnych stworzono funkcję *SwapLastAndIndex*, dzięki której nie jest potrzebne tworzenie kopii listy nieodwiedzonych wierzchołków, lecz zamieniania jest tylko kolejność wierzchołków oraz następnie przekazywana jest tylko część tablicy (wykorzystywane są tutaj operacje na slice'ach w Go).

```
// SwapLastAndIndex Swaps last and the element on the specified inex
func SwapLastAndIndex(slice []int, index int) {
        if len(slice) > index {
                replaced := slice[len(slice)-1]
                slice[len(slice)-1] = slice[index]
                slice[index] = replaced
```

```
        }
}
```

Jeśli zostały odwiedzone już wszystkie wierzchołki to sprawdzamy, czy obecna ścieżka jest lepsza od dotychczas znalezionej. W tym celu stworzono funkcję *TargetFunction*, która oblicza koszt podanej ścieżki. Jej implementację przedstawiono poniżej.

```
// TargetFunction returns total cost of given path in given adjacencyMatrix
func (b *BruteForce) TargetFunction(nodes []int) int {

        var result int
        last := nodes[0]
        for _, node := range nodes[1:] {
                result = result + b.adjacencyMatrix[last][node]
                last = node
        }

        result = result + b.adjacencyMatrix[nodes[len(nodes)-1]][nodes[0]]
        return result
}
```

2.3 Metoda podziału i ograniczeń - Branch and Bound

Metoda podziału i ograniczeń polega na przeszukiwaniu drzewa reprezentującego przestrzeń rozwiązań problemu. Przeszukanie całego drzewa byłoby bardzo kosztowne dlatego metoda ta wykorzystuje ograniczenia (ang. *bound*), które pozwalają określić daną ścieżkę jako obiecującą, lub nie. W dalszej fazie algorytm przegląda tylko potomków węzłów obiecujących.Pozwala to, razem z dobraniem odpowiedniej strategii odwiedzania wierzchołków (ang. *branch*) oraz liczenia granicy, zmniejszyć ilość odwiedzonych wierzchołków i szybciej znaleźć rozwiązanie problemu.

Wybrane strategie przeszukiwania drzewa

Podczas przeszukiwania drzewa rozwiązań można wybrać jedną z dwóch metod przeszukiwania - wgłąb oraz wszerz. Podczas przeszukiwania wszerz w każdym kroku przeglądamy wszystkie wierzchołki na wybranym poziomie n drzewa, następnie wszystkie na poziomie n+1 i dalej aż do liści. Podczas implementacji algorytmu Branch and Bound zdecydowano się na użycie przeszukiwania wgłąb. W tej strategii po przejrzeniu synów wybranego węzła przeglądamy wszystkie pozostałe nierozwinięte obiecujące węzły i rozwijamy węzeł o najlepszej granicy. Pozwala to na szybsze lepsze oszacowanie górnej granicy (ang. *upper-bound*).

Wybrane strategie ograniczania drzewa przeszukiwań

Do zmniejszenia drzewa przeszukiwań określa się dwa parametry - górną oraz dolną granicę (ang. *upper and lower bound*). W przypadku problemu minimalizacyjnego jakim jest problem komiwojażera granica górna mówi nam o maksymalnym możliwym rozwiązaniu dla przeglądanego wierzchołka, a dolna granica określa odpowiednio minimalne rozwiązanie.
W każdym odwiedzanym węźle szacowana jest wartość dolnego ograniczenia, która musi być nie większa niż wartość aktualnego górnego ograniczenia. Jeśli obliczona wartość dolnego ograniczenia jest większa niż wartość górnego ograniczenia to wierzchołek ten nie jest przeglądany dalej, zostając uznany za nieobiecujący.

Implementacja algorytmu - o zastosowanych rozwiązaniach

Metoda rozgałęziania: przeszukiwanie wgłąb
Szacowanie górnej granicy: pierwsze najlepsze rozwiązanie
Szacowanie dolnej granicy:
Początkowo dolna granica dla grafu obliczana jest sumując dwie minimalne wagi wierzchołków dla każdego z wierszy macierzy sąsiedztwa. Wartości te są sumowane i dzielone przez 2. W każdym kolejnym kroku rekurencyjnego wywołania funkcji przeszukującej od aktualnej dolnej granicy odejmujemy sumę pierwszego i/lub drugą minimalną wartość z wiersza odwiedzanego wierzchołka podzieloną przez dwa (na pierwszym poziomie grafu każdy z wierzchołków odwiedzamy po raz pierwszy, więc usuwamy sumę wag dwóch pierwszych minimalnych wartości z wierzchołka początkowego i drugiego w kolejności. Na kolejnych poziomach usuwamy sumę wag drugiego minimalnego ostatnio odwiedzonego wierzchołka i pierwszą minimalną nowo odwiedzanego podzielonego przez dwa). Następnie dolną granicę powiększamy o rzeczywisty koszt przejścia pomiędzy ostatnim wierzchołkiem i nowo odwiedzonym. W kolejnym kroku sprawdzamy, czy otrzymana dolna granica jest mniejsza lub równa aktualnej górnej granicy (*upperBound*) - jeśli tak to wierzchołek ten jest przeszukiwany wgłąb. W przeciwnym wypadku wierzchołek ten jest opuszczany jako nieobiecujący.
Podczas testowania algorytmu zauważono jednak, że przyjęta w ten sposób dolna granica jest oszacowaniem niewłaściwym dającym zbyt pesymistyczne oszacowania dla niektórych wierzchołków przez to dla zbiorów dwóch zbiorów danych dawała niepoprawne wyniki, ponieważ ścieżka która mogła być optymalnym rozwiązaniem otrzymywała oszacowanie powyżej aktualnej górnej granicy. W celu naprawienia algorytmu przyjęto nowe początkowe oszacowanie dla dolnej granicy - suma wag dwóch minimalnych krawędzi dla każdego wierzchołka dzielona jest teraz przez 3, a nie przez 2 [1]. Oszacowanie to nie jest tak dokładne, lecz wystarczające żeby odrzucać na tyle dużą liczbę krawędzi, że algorytm nadal szybko rozwiązuje problemy o rozmiarze 20.

Implementacja algorytmu - działanie algorytmu

W pierwszym etapie działania algorytmu szacowana jak początkowa dolna granica rozwiązania. Funkcję do szacowania pierwszej dolnej granicy rozdzielono na mniejsze funkcje dla czytelności.

```
func (b *BranchAndBound) calculateFirstLowerBound() int {
        lowerBound := 0
        for row := 0; row < b.size; row++ {
                lowerBound += b.findFirstMin(row)
                lowerBound += b.findSecondMin(row)
        }

        return lowerBound / 3
}

func (b BranchAndBound) findFirstMin(row int) int {
        min := math.MaxInt64

        for column := 0; column < b.size; column++ {
                if b.adjacencyMatrix[row][column] < min
                    && b.adjacencyMatrix[row][column] != -1 {
                        min = b.adjacencyMatrix[row][column]
                }
```

[1] Źródło, gdzie znalazłam niepoprawną metodę obliczania dolnego ograniczenia : `https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/`

```
        }

        return min
}

func (b BranchAndBound) findSecondMin(row int) int {
        min1 := b.findFirstMin(row)
        isMin1AlreadyFound := false
        min2 := math.MaxInt64

        for column := 0; column < b.size; column++ {
                currentValue := b.adjacencyMatrix[row][column]

                if currentValue <= min2 && currentValue != -1 {
                        if currentValue == min1 {
                                if !isMin1AlreadyFound {
                                        isMin1AlreadyFound = true
                                } else {
                                        return min1
                                }
                        } else {
                                min2 = currentValue
                        }
                }
        }

        return min2
}
```

Funkcja obliczająca optymalną ścieżkę jako parametry przyjmuje aktualne oszacowanie dolnej granicy ($lowerBound$), tablicę zawierającą aktualną ścieżkę ($path$), mapę (odpowiednik słownika w Golangu) zawierającą informacje, czy dany wierzchołek był już odwiedzony ($isVisited$) oraz koszt dotychczasowo zbudowanej ścieżki. Pierwsze wywołanie funkcji przedstawiono poniżej:

```
        for i := 0; i < b.size; i++ {
                isVisited[i] = false
        }

        isVisited[STARTING_NODE] = true
        firstLower := b.calculateFirstLowerBound()
        b.evaluateRecursively(firstLower, []int{STARTING_NODE}, isVisited, 0)
```

Pierwsze dolne ograniczenie obliczane jest na podstawie algorytmu wyjaśnionego powyżej i implementacji przedstawionej w poprzednim listingu. Jako startowy wierzchołek przyjmowana jest stała $STARTING_NODE$, a jako mapa zawierająca informacje, czy dane wierzchołek został odwiedzony inicjalizowana jest z wartościami $false$ poza startowym wierzchołkiem. Początkowy koszt dotychczasowej ścieżki wynosi 0.
Funkcja obliczająca rekurencyjnie optymalną ścieżkę w każdym wywołaniu sprawdza, czy dotychczasowo zbudowana ścieżka równa jest wielkości grafu - jeśli tak oznacz to, że został stworzony pełny cykl i pozostało sprawdzić, czy otrzymane rozwiązanie jest lepsze od aktualnego najlepszego. W tym celu sprawdzane jest, czy aktualny koszt($cost$) powiększony o koszt powrotu do miasta początkowego jest większy od aktualnej górnego ograniczenia. Jeśli otrzymany koszt jest mniejszy, znalezione zostało nowe, lepsze

rozwiązanie, zmieniana jest aktualne górne ograniczenie, a ścieżka zapamiętywana. Jeśli rozmiar zbudowanej dotychczas ścieżki jest mniejszy od rozmiaru grafu to dla każdego nieodwiedzonego wierzchołka grafu tworzona jest ścieżka z wierzchołkiem dodanym do aktualnie zbudowanej ścieżki *path*. Następnie szacowane jest dolne ograniczenie dla nowej ścieżki według algorytmu opisanego wyżej w punkcie "Implementacja algorytmu - o zastosowanych rozwiązaniach" i sprawdzane jest,czy warto dalej rozwijać daną ścieżkę porównując otrzymane dolne ograniczenie do aktualnego górnego ograniczenia.

```
func (b *BranchAndBound) evaluateRecursively
    (lowerBound int, path []int, isVisited map[int]bool, cost int) {

        if len(path) == b.size {
                cost += b.adjacencyMatrix[path[len(path)-1]][0]
                if cost < b.upperBound {
                        b.bestPath = make([]int, len(path))
                        copy(b.bestPath, path)
                        b.upperBound = cost
                }

                return
        }

        for i := 0; i < b.size; i++ {

                if !isVisited[i] {

                        newBound := lowerBound
                        newCost := cost

                        if len(path) == 1 {
                                newBound -=
                                    (b.findFirstMin(0) + b.findFirstMin(i)) / 2
                        } else {
                                newBound -=
                                    (b.findSecondMin(path[len(path)-1]) + b.findFirstMin(i)) / 2
                        }

                        isVisited[i] = true
                        newCost += b.adjacencyMatrix[path[len(path)-1]][i]

                        if newBound+newCost <= b.upperBound {
                                newPath := append(path, i)
                                b.evaluateRecursively(newBound, newPath, isVisited, newCost)
                        }


                        isVisited[i] = false
                }
        }
}
```

2.4 Programowanie dynamiczne - algorytm Helda-Karpa

Programowanie dynamiczne

Alogrytm Helda-Karpa wykorzystuje programowanie dynamiczne do rozwiązania problemu komiwojażera. Dana instancja problemu jest dzielona na mniejsze podproblemy i rozwiązanie jest otrzymywane bazując na rozwiązaniach mniejszych podproblemów. Podejście to może przypominać strategię "dziel i zwyciężaj", jednak w programowaniu dynamicznym tworzone podproblemy są zależne od siebie, co w strategii "dziel i zwyciężaj" nie występuje - mniejsze podproblemy są od siebie niezależne.

Działanie algorytmu

Załóżmy, że mamy n wierzchołków ponumerowanych kolejno 1,2, .., n. Jako $d_{i,j}$ oznaczmy odległość między wierzchołkami. Niech wierzchołek 1 będzie punktem początkowym.
Oznaczymy jako $D(S, p)$ optymalną długość ścieżki z punktu 1 do punktu p, przechodzącą przez wszystkie wierzchołki w zbiorze S, przy czym $p \in S$.
Stąd na przykład:

- Jeśli $|S| = 1$, to $D(S, p) = d_{1,p}$
- Jeśli $|S| > 1$, to $D(S, p) = min_{x \in (S-\{p\})}(D(S - \{p\}, x) + d_{x,p})$

W rzeczywistości w każdym rozwiązaniu podproblemu D należy ustalić, który punkt powinien być przedostatni w trasie (jaki punkt ma poprzedzać p).

Implementacja algorytmu

Do przechowywania podrozwiązań problemów użyto struktury *map*, która w Golangu jest odpowiednikiem słownika. Kluczem tej struktury była para zmiennych typu *int* przechowującej ostatni odwiedzonej wierzchołek (p w opisie algorytmu powyżej) oraz listę odwiedzonych już miast. Zmienna typu całkowitego zawierająca listę odwiedzonych już miast była generowana na podstawie zmiany wartości poszczególnych bitów - n-ty bit zmiennej odpowiadał stanowi odwiedzenia danego wierzchołka (1 - odwiedzone, 0 - nieodwiedzone). Wartością przechowywaną w słowniku była para liczb całkowitych przechowująca koszt rozwiązania danego podproblemu oraz ostatni odwiedzonych wierzchołek. Struktury te można zobaczyć w listingu poniżej.

```
// PartialSolution defines value stored in partialSolutions
// map of HeldKarp algorithm
type PartialSolution struct {
        cost     int
        lastNode int
}

// Key defines identifier for PartialSolution
// in partialSolutions map of HeldKarp algorithm
type Key struct {
        lastNode     int
        visitedNodes int
}
```

W celu rozwiązania wszystkich podproblemów stworzono funkcję rekurencyjną przyjmującą jako parametry listę nieodwiedzonych wierzchołków ($nodesToVisit$) oraz poprzednio użyty klucz w słowniku podrozwiązań ($lastSolutionKey$). W każdym wywołaniu funkcji sprawdzamy, czy zostały jeszcze jakieś nieodwiedzone wierzchołki na liście $nodesToVisit$. Jeśli lista ta jest pusta to oznacza to, że w kroku

tym wracamy do wierzchołka początkowego, więc do kosztu tego rozwiązania należy przypisać koszt przejścia z wierzchołka *destination* do wierzchołka początkowego. Jeśli lista ta nie jest pusta szukamy najlepszego rozwiązania wśród nieodwiedzonych jeszcze wierzchołków. W tym celu iterujemy po pętli nieodwiedzonych wierzchołków, a w każdej pętli generujemy nowy klucz do słownika podrozwiązań - jeśli taki klucz już istnieje, pobieramy tą wartość i porównujemy z obecnie najlepszym znalezionym rozwiązaniem. Natomiast jeśli dany klucz nie istnieje tworzymy nowe rekurencyjne wywołanie funkcji w celu znalezienia rozwiązania podproblemu z listą $nodesToVist$ bez wybranego wierzchołka z pętli oraz z $lastSolutionKey$ zawierającymi wierzchołek z pętli jako ostatnio odwiedzony wierzchołek oraz klucz w postaci liczby całkowitej z bitami odwiedzonych wierzchołków ustawionych na wartość 1.

```
func (h *HeldKarp) calculatePaths(nodesToVisit []int, lastSolutionKey Key) PartialSolution
{

        if len(nodesToVisit) <= 0 {
                var newPartialSolution PartialSolution
                newPartialSolution.cost =
                    h.adjacencyMatrix[lastSolutionKey.lastNode][h.startingNode]
                newPartialSolution.lastNode =
                    lastSolutionKey.lastNode

                return newPartialSolution
        }

        currentBestSolution := PartialSolution{math.MaxInt64, lastSolutionKey.lastNode}
        iterationSolution := PartialSolution{}

        for index, node := range nodesToVisit {
                keyWithNewNode := Key{node, setBit(lastSolutionKey.visitedNodes, node)}
                partialSolution, keyExists := h.partialSolutions[keyWithNewNode]

                if !keyExists {
                        SwapLastAndIndex(nodesToVisit, index)
                        partialSolution =
                            h.calculatePaths
                                (nodesToVisit[:len(nodesToVisit)-1], keyWithNewNode)
                        SwapLastAndIndex(nodesToVisit, index)
                        if len(nodesToVisit) > 1 {
                                h.partialSolutions[keyWithNewNode] = partialSolution
                        }
                }
                iterationSolution.cost
                    = partialSolution.cost
                        + h.adjacencyMatrix[lastSolutionKey.lastNode][node]
                iterationSolution.lastNode
                    = node

                if iterationSolution.cost < currentBestSolution.cost {
                        currentBestSolution = iterationSolution
                }
        }
```

```
        return currentBestSolution
}
```

Pierwsze wywołanie rekurencyjnej funkcji odbywa się poprzez przekazanie listy wszystkich wierzchołków grafu poza wierzchołkiem początkowym (domyślnie uznanym jako wierzchołek zerowy), a klucz ostatniego rozwiązania to para liczb całkowitych, gdzie ostatnio odwiedzony wierzchołek *lastNode* to wierzchołek początkowy, a klucz odzwierciedlający odwiedzeniu tylko zerowego wierzchołka (domyślnie) to 1 (tylko zerowy bit ustawiamy na wartość 1).

```
h.calculatePaths(nodes, Key{h.startingNode, 1})
```

Funkcja wyszukująca rekurencyjnie najlepsze rozwiązania zwraca klucz do wartości przechowującej tylko jeden wierzchołek jaki należy odwiedzić po startowym. W tym celu stworzono funkcję *backtrackOptimalPath*, która na podstawie otrzymanego klucza wierzchołkiem do odwiedzenia po startowym sprawdza wartości przechowywane w słowniku i odtwarza pełną, optymalną ścieżkę. W każdym kroku funkcja ta tworzy klucz kolejnego rozwiązanego podproblemu w którym przechowywana jest informacja o ostatnio odwiedzonym wierzchołku zmieniającej poszczególne bity w zmiennej całkowitej przechowującej odwiedzone już wierzchołki.

```
func (h *HeldKarp) backtrackOptimalPath(lastNode int) []int {
        var optimalPath []int
        optimalPath = append(optimalPath, h.startingNode)
        optimalPath = append(optimalPath, lastNode)
        lastVisitedNodesKey := setBit(1, lastNode)
        lastKey := Key{lastNode, lastVisitedNodesKey}
        partialSolution := h.partialSolutions[lastKey]

        for len(optimalPath) < len(h.adjacencyMatrix[0]) {
                optimalPath = append(optimalPath, partialSolution.lastNode)
                lastVisitedNodesKey =
                    setBit(lastVisitedNodesKey, partialSolution.lastNode)
                lastKey := Key{partialSolution.lastNode, lastVisitedNodesKey}
                partialSolution = h.partialSolutions[lastKey]
        }

        return optimalPath
}
```

3 Eksperymenty obliczeniowe

Obliczenia zastały wykonane na komputerze klasy PC z procesorem i5-5200U, kartą graficzną NVIDIA GeForce GTX 940M, 12GB RAM i DYSK SSD. Jako miarę jakości algorytmu przyjęto uśredniony czas (pomiary powtórzono 10-krotnie) rozwiązania danej instancji problemu zmierzony w milisekundach. Wszystkie wyniki zebrano i przedstawiono w tabeli nr 1 gdzie:

- n - rozmiar problemu (ilość wierzchołków),
- t_{BF} - średni czas dla algorytmu Brute Force,
- t_{BnB} - czas dla algorytmu Branch and Bound,
- t_{HK} - czas dla algorytmu Helda-Karpa

Nazwa pliku	n	t_{BF} **(ms)**	t_{BnB} *(ms)*	t_{HK} *(ms)*
data10	10	585000	10986	1001
data11	11	3671995	45016	2988
data12	12	29422016	487024	9004
data13	13	367627825	2968981	31996
data14	14		653000	50004
data15	15		47432019	138999
data16	16			374991
data17	17			818999
data18	18			2406994
data21	21			23334021

Table 1: Czas obliczeń w milisekundach dla instancji o rozmiarze n.

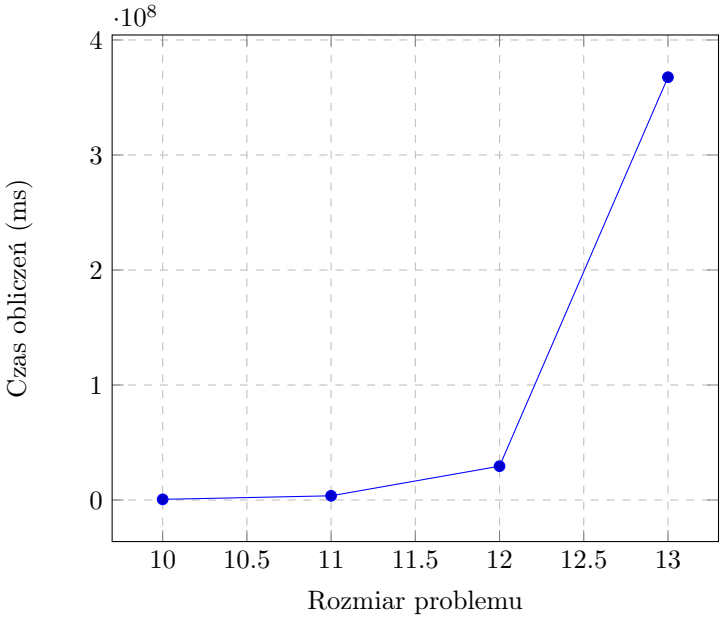

Figure 1: Czas obliczeń w zależności od rozmiaru problemu dla algorytmu BruteForce.

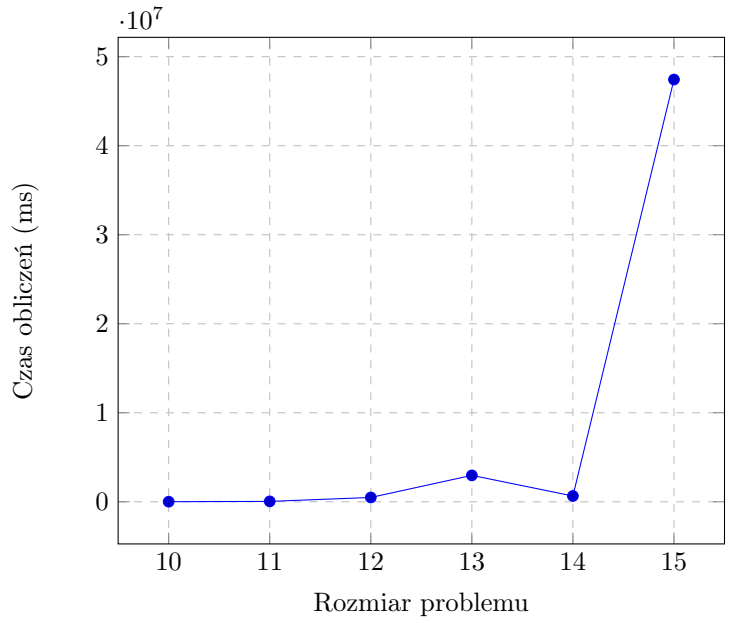

Figure 2: Czas obliczeń w zależności od rozmiaru problemu dla algorytmu Branch and Bound.

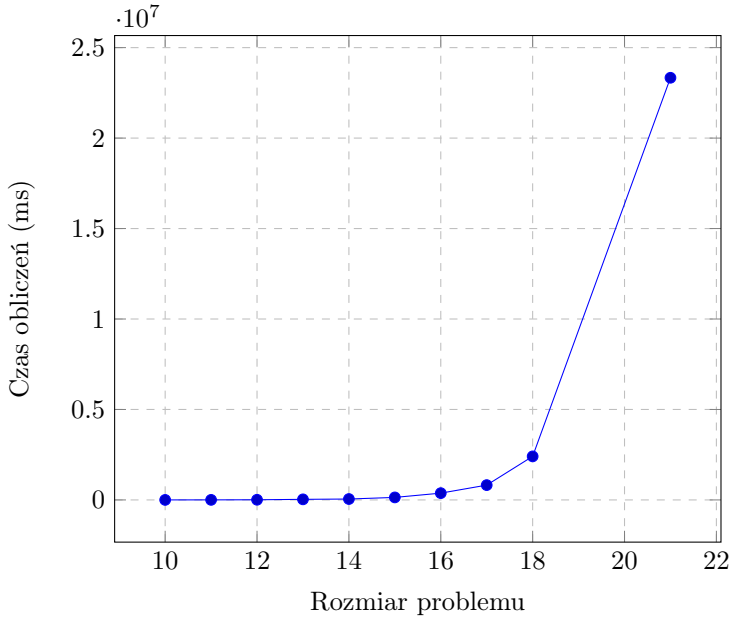

Figure 3: Czas obliczeń w zależności od rozmiaru problemu dla algorytmu Helda-Karpa.

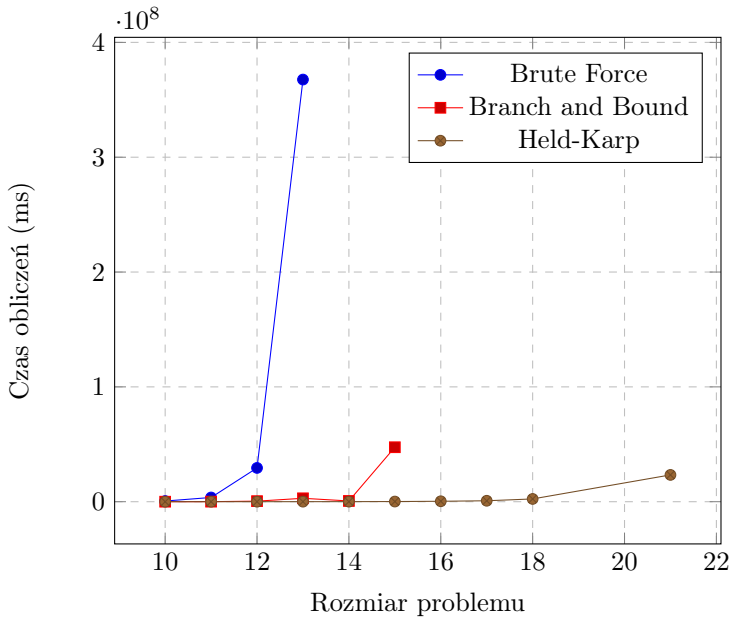

Figure 4: Czas obliczeń w zależności od rozmiaru problemu - porównanie

4 Wnioski

Problem komiwojażera to problem NP-zupełny, czyli taki, którego nie da się rozwiązać w czasie wielomianowym. Dokonanie przeglądu zupełnego tego problemu złożoności obliczeniowej daje nam złożoność obliczeniową $O(n!)$. Można jednak wykorzystywać różne metody zmniejszania przestrzeni stanów wykorzystując algorytm Branch And Bound, którego wydajność może zależeć od dobranej metody wyznaczania górnego i dolnego ograniczenia. Alternatywą dla programowania zachłannego może być też programowanie dynamiczne i algorytm Helda-Karpa, który ma złożoność obliczeniową $0(n^2 2^n)$.
Na rysunku 4. przedstawiającym porównanie czasu wykonania poszczególnych algorytmów można zaobserwować bardzo szybki wzrost czasu obliczeń dla algorytmu Brute Force. Najlepiej sprawdza się algorytm Helda-Karpa, który nawet dla największej testowej instancji działa w średnim czasie znacznie poniżej minuty (w przybliżeniu 24 sekundy). Z rysunku 2 można wywnioskować, że algorytm Branch and Bound jest najmniej przewidywalny, ponieważ zależnie od rozłożenia kosztów w określonym zbiorze danych wiele ścieżek może być porzucanych jako nieopłacalne lub koszty przejścia każdej z nich mogą być zbliżone (w tym przypadku niewiele ścieżek może być odrzucone na wczesnym etapie jako nieopłacalne). W otrzymanych zbiorach testowych dane o rozmiarze 13 (plik data13) są rozłożone bardziej równomiernie niż w zbiorze danych o rozmiarze 14 (plik data14). W rezultacie algorytm dla danych o rozmiarze 14. jest w stanie wyeliminować więcej ścieżek, więc czas wykonania jest mniejszy niż dla danych o rozmiarze 13.

5 Bibliografia

- https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/
- http://translogistics.pl/files/jtl/2015/R1.pdf
- http://algorytmy.ency.pl/artykul/problem_komiwojazera
- https://eduinf.waw.pl/inf/alg/001_search/0140.php
- http://algorytmy.ency.pl/artykul/algorytm_helda_karpa
- https://www.ii.uni.wroc.pl/~prz/2011lato/ah/opracowania/met_podz_ogr.opr.pdf
- http://www.cs.put.poznan.pl/mkasprzak/zp/ZP-wyklad5.pdf
- https://www.princeton.edu/~rblee/ELE572Papers/p137-sedgewick.pdf?fbclid=IwAR2oEYzqRUD-w4vfDOY_j8F2xJt-Ynm5fpO7WTUsNM4v1nr4IApjnOSNAlU

Contents