Łukasz Lepak, 277324

Analiza algorytmów
Dokumentacja końcowa

prowadzący: dr inż. Tomasz Gambin

1. Treść projektu

Dane jest N punktów rozrzuconych na płaszczyźnie o zadanych współrzędnych (x, y). Należy skonstruować algorytm, który poprzez odpowiednie połączenie punktów utworzy spiralę, która będzie łączyć wszystkie punkty oraz będzie skręcona tylko w jedną stronę (brak kątów większych niż 180 stopni po wewnętrznej stronie spirali). Porównać czas obliczeń i wyniki różnych metod.

2. Założenia

2.1. Odstępstwa od koncepcji wstępnej

a) Dwa algorytmy podane we wstępnej koncepcji (algorytm sortowania punktów oraz algorytm zasięgu promieni) nie zostały zaimplementowane ze względu na ich wcześniej niezauważoną niepoprawność i generowanie złych rozwiązań.
b) Dodano jeden nowy algorytm – algorytm rosnących promieni. Jego opis znajduje się poniżej.
c) Zmodyfikowano jeden z algorytmów znajdujących się w dokumentacji wstępnej – algorytm otoczek wypukłych. Zmieniono sposób łączenia ze sobą otoczek. Zostanie on opisany niżej.
d) Zrezygnowano z implementowania opcji GUI użytkownika na rzecz uruchamiania aplikacji z odpowiednimi komendami i parametrami z konsoli. Miało to na celu uproszczenie struktury programu oraz bardziej przejrzysty sposób dostosowywania opcji aplikacji. Możliwe wywołania są opisane w pliku *readme.txt* oraz poniżej. GUI do wyświetlania wyników obliczeń pozostało bez zmian.
e) Zrezygnowano z pisania testów jednostkowych aplikacji – kolejność punktów na spirali, choć generuje sie w sposób deterministyczny, to jest trudna do obliczenia, przez co testy byłyby możliwe tylko dla małych rozmiarów danych wejściowych, dla których rezultaty są widoczne gołym okiem w oknie aplikacji. Dla większych danych wejściowych rezultaty również można ocenić patrząc na

wygenerowaną spiralę w oknie aplikacji.
f) Początek układu współrzędnych został ustawiony w lewym górnym rogu ekranu - dzięki temu jest on zgodny z obiektami graficznymi JavaFX.

2.2. Dodatkowe założenia

a) Generator danych losowych generuje dane w zakresie szerokości – dla współrzędnej x – oraz wysokości – dla współrzędnej y – ekranu. Dzięki temu uzyskane wyniki można łatwo przenieść na wizualizację za pomocą obiektów graficznych JavaFX.
b) Przed pomiarami czasu wykonania programu wykonywany jest *warm-up* maszyny wirtualnej Javy w celu osiągnięcia jak najmniejszych opóźnień w kompilacji oraz działaniu algorytmów.
c) Jeśli program otrzyma złą kombinację parametrów wejściowych przy uruchamianiu to poinformuje o tym fakcie użytkownika i wydrukuje na standardowym wyjściu pomoc ze wszystkimi możliwymi opcjami wywołania programu.
d) W trybie wywołania -m1, jeśli program otrzyma błędne dane to poinformuje o tym użytkownika i zakończy się.
e) W algorytmie rosnących promieni została zdefiniowana stała ilość parametru *slice* – ilości części, które składają się na cały okrąg.
f) Punktem środkowym dla algorytmu rosnących promieni jest środek okna, na którym rysowana jest spirala. Dzięki temu wizualizowany efekt zajmuje centralną część ekranu.
g) Punkty – po wyliczeniu przez algorytm rosnących promieni – są normalizowane przed narysowaniem w celu zmieszczenia ich na wymiarach okna.
i) Połączenia między punktami realizowane są liniami prostymi.

3. Zaimplementowane algorytmy

3.1. Algorytm otoczek wypukłych (Convex Hull Algorithm)

3.1.1. Procedura ogólna

1. dopóki liczba punktów do rozpatrzenia nie mniejsza niż trzy
    1.1. wyznacz otoczkę wypukłą z dostępnych punktów, dodaj ją na stos otoczek
2. jeśli zostały jakieś punkty to dodaj je do jednego zbioru i dodaj na stos otoczek
3. dopóki stos otoczek nie jest pusty
    3.1. zdejmij ze stosu nową otoczkę do rozpatrzenia

3.2. jeśli w spirali są już co najmniej 2 punkty

3.2.1. wyznacz punkt na otoczce, z którym przecina się prosta utworzona z dwóch ostatnio dodanych punktów spirali

3.2.2. dodaj ten punkt do spirali

3.3. dodaj punkty z otoczki do spirali

3.1.2. Opis algorytmu

Na początku działania algorytmu wyznaczane są otoczki wypukłe dla pomniejszającego się zbioru punktów. Otoczki wypukłe są wyznaczane za pomocą algorytmu Grahama. Korzysta on z sortowania punktów względem kąta utworzonego z punktem o najmniejszej współrzędnej y oraz z funkcji określającej, po której stronie prostej określonej przez 2 punkty znajduje się trzeci punkt. Otoczki są wyznaczane i umieszczane na stosie aż do momentu, w którym liczność początkowego zbioru punktów będzie mniejsza niż 3, ponieważ z takiej ilości punktów nie można utworzyć kolejnej otoczki. Pozostałe punkty również trafiają do jednego stosu, który ostatecznie ląduje na szczycie stosu z otoczkami. Następnie, zdejmując kolejne stosy punktów ze stosu otoczek, konstruujemy spiralę. Połączenie otoczek realizujemy poprzez znalezienie punktu przecięcia prostej stworzonej z dwóch ostatnich punktów na spirali oraz odcinka złożonego z dwóch punktów na otoczce „wyższej". Takie przecięcie może nastąpić tylko z jednym takim odcinkiem, ponieważ rozpatrujemy tylko przecięcia, które nastąpiły poprzez przedłużenie prostej od przedostatniego punktu przez ostatni i dalej. W poszukiwaniu punktu przecięcia uwzględnione są różne możliwe przypadki, takie jak, na przykład, prosta pionowa, która nie ma jednoznacznego równania liniowego, czy proste równoległe. Taki punkt przecięcia jest dodawany do spirali. Procedura ta powtarza się, aż nie dojdziemy do ostatniej otoczki, która kończy naszą spiralę.

3.1.3. Analiza złożoności algorytmu

Algorytm Grahama wyznaczania otoczki wypukłej ma złożoność *O(n * log(n)).* Wynika to z sortowania, które jest przeprowadzanie podczas wykonywania tej procedury. W tym algorytmie wyznaczanych jest wiele takich otoczek. Pesymistyczny przypadek przewiduje, że wykonamy n/3 takich sortowań na zmniejszających się zbiorach danych – za każdym razem wyznaczona otoczka będzie trójkątem. Można wykazać, że:

$$\lim_{n\to\infty} \sum_{k=1}^{\frac{n}{3}} 3k * \log(3k) \ \in O(n^2 * \log(n))$$

Z tej zależności wynika, że procedura wyznaczania otoczek wypukłych w pesymistycznym przypadku daje nam złożoność czasową *O($n^2 * \log(n)$).* W przypadku tego algorytmu jest to czynnik dominujący – w dalszej części algorytmu w pesymistycznym przypadku wykonamy maksymalnie *2 * n* operacji o wadze jednostkowej, co nie ma wpływu na ostateczną złożoność algorytmu. Zatem algorytm otoczek wypukłych ma złożoność obliczeniową *O($n^2 * \log(n)$).*

3.2. Algorytm rosnących promieni (Rising Radius Algorithm)

3.2.1. Procedura ogólna

1. posortuj punkty według rosnącej odległości od punktu środkowego O
2. dodaj pierwszy punkt z listy do spirali, usuń go z listy i ustaw jako punkt poprzedni
3. dopóki wszystkie punkty nie zostały dodane do spirali
    3.1. weź punkt aktualny jako pierwszy z listy i usuń go z listy
    3.2. wyznacz różnicę między promieniami dwóch kolejnych punktów
    3.3. wyznacz różnicę między kątami dwóch kolejnych punktów
    3.4. jeśli różnica jest większa niż 180 stopni
        3.4.1. zwiększ ilość rysowanych części okręgu o *slice* (pełny okrąg)
    3.4. dodawaj do spirali punkty leżące na krzywej między dwoma punktami o liniowo rosnącym promieniu i kącie
    3.5. dodaj punkt aktualny do spirali i ustaw punkt poprzedni jako aktualny

3.2.2. Opis algorytmu

Algorytm opiera się na odległościach danych punktów od pewnego zdefiniowanego punktu środkowego oraz na interpolacji wartości pośrednich między dwoma punktami. Przed uruchomieniem algorytmu musi być określony punkt środkowy, wokół którego będziemy sortować punkty oraz wielkość parametru *slice*, który decyduje o tym, jak gładką spiralę otrzymamy. Na początku działania algorytmu sortujemy punkty względem ich odległości od wybranego punktu środkowego. Pierwszy punkt na posortowanej liście automatycznie trafia do spirali i zostaje oznaczony jako punkt początkowy. Następnie bierzemy kolejny punkt z listy, oznaczamy go jako punkt aktualny i wyznaczamy różnicę w promieniach i kątach między tymi punktami. Obliczamy ilość punktów do dodania między tymi punktami. Następnie punkty te generujemy poprzez interpolację liniową promienia oraz kątów. Każdy kolejny punkt ma trochę większy kąt oraz trochę większy promień od poprzedniego. Wszystkie te punkty dodajemy do spirali. Po wygenerowaniu wszystkich punktów pośrednich dodajemy punkt aktualny do spirali, usuwamy go z listy

punktów, oraz ustawiamy punkt początkowy jako punkt aktualny. Czynności od pobierania kolejnego punktu z listy powtarzamy, aż z posortowanej listy nie znikną wszystkie punkty i nie znajdą się na spirali. Warunek dodania dodatkowego okręgu jeśli dwa kolejne punkty są oddalone od siebie o więcej niż 180 stopni ma zapobiec przypadkowi utworzenia się w spirali kątów większych niż 180 stopni.

3.2.3. Analiza złożoności algorytmu

Na początku algorytmu wykonywane jest sortowanie punktów, które posiada złożoność czasową *O(n * log(n))*. Złożoność czasowa generowania nowych punktów jest liniowa (*O(n)).* Należy jednak zwrócić uwagę, złożoność ta w pesymistycznym przypadku może mieć duży współczynnik stały (równy 1,5 * *slices)*, co może spowodować, że dla mniejszych n to ta złożoność będzie miała większy wpływ na czas wykonywania algorytmu. Niemniej asymptotycznie większe oddziaływanie na długość obliczeń ma złożoność sortowania, zatem algorytm ma pesymistyczną złożoność *O(n * log(n))*.

4. Opis kompilacji i uruchomienia programu

Podobny opis znajduje się w pliku *readme.txt.*

4.1. Kompilacja

Program wymaga do kompilacji zainstalowanego środowiska Java. Przed kompilacją należy przejść do katalogu głównego z plikami źródłowymi programu. Kompilację przeprowadzamy komendą:
javac AAL/AAL.main
Komenda ta utworzy wymagane pliki *.class*, które później możemy uruchomić.

4.2. Uruchomienie programu

Program możemy uruchomić na 4 możliwe sposoby:
1. java AAL.AAL -m1 -a{algorytm}
2. java AAL.AAL -m2 -a{algorytm} -n{ilość_punktów} (-si) (-so)
3. java AAL.AAL -m3 -n{ilość_punktów} -s{skok} -t{ilość_testów} -i{iteracje}
4. java AAL.AAL -help

Opis metod uruchomienia programu:
1. Wywołuje aplikacje w trybie czytania ze standardowego wejścia oraz z wybranym algorytmem do wykonania obliczeń. Wyniki prezentuje graficznie oraz poprzez wypisanie ich na standardowe wyjście. Dzięki temu możliwe jest przekierowanie strumieni wejściowych i wyjściowych do pliku.

2. Wywołuje aplikacje w trybie losowania danych wejściowych o zdefiniowanej ilości oraz z wybranym algorytmem do wykonania obliczeń. Wyniki prezentuje graficznie. Pozwala zdefiniować opcje do zapisania danych wejściowych lub/i wyjściowych na standardowe wyjście.
3. Uruchamia aplikacje w trybie mierzenia czasu działania algorytmów oraz sprawdzania ich złożoności obliczeniowej. Obliczenia zaczynają się od zdefiniowanej wielkości danych, zwiększają się o zdefiniowany skok, wykonuje się podana ilość testów, a czas obliczeń dla określonej wielkości danych wejściowych jest uśredniany na podstawie ilości iteracji. Wynikiem działania tego trybu jest wydruk na standardowe wyjście określający dla wszystkich algorytmów wielkość danych wejściowych, średni czas wykonania w milisekundach oraz wartość współczynnika q.
4. Wypisuje na standardowe wyjście możliwe opcje wywołania programu.

Opis parametrów:
-m1, -m2, -m3, -help - wybór opcji wykonania programu.
-a{algorytm} - algorytm, który zostanie użyty do wykonania obliczeń, zmienna {algorytm} może przyjmować wartości CH lub RR. (CH - metoda otoczek wypukłych, RR - metoda rosnących promieni)
-n{ilośc_punktów} - definiuje wielkość danych wejściowych losowanych (dla trybu -m2) bądź wielkość danych startowych (dla -m3). Wartość zmiennej {ilośc_punktów} powinna być liczbą całkowitą nie mniejszą niż 3.
-s{skok} - definiuje wielkość skoku między wielkością danych wejściowych w trybie wywołania -m3. Zmienna {step} powinna być nieujemną liczbą całkowitą.
-t{ilość_testów} - definiuje ilość różnych wielkości danych wejściowych, dla których wykonają się testy. Zmienna {ilość_testów} powinna być liczbą nieparzystą nie mniejszą niż 1. Liczby parzyste zostaną zwiększone w programie o 1.
-i{iteracje} - ilość iteracji, po której będzie uśredniany czas działania algorytmu dla trybu wywołania -m3. Zmienna {iteracje} powinna być dodatnią liczbą całkowitą.
-si, -so - opcjonalne parametry definiujące, czy tryb wywołania -m2 będzie wypisywał wygenerowane wejście i/lub obliczone wyjście na standardowe wyjście.

5. Opis plików

Pliki w ramach aplikacji zostały podzielone na pakiety:
a) AAL – pakiet główny aplikacji zawierający plik z metodą main:
-AAL.java – plik, od którego zaczyna się wykonanie programu, przesyła argumenty wywołania do dalszych struktur danych
b) algorithms – pakiet zawierający definicje klas odpowiedzialnych za generowanie spiral na podstawie algorytmów:
-ConvexHullAlgorithm.java – algorytm otoczek wypukłych
-RisingRadiusAlgorithm.java – algorytm rosnących promieni
-SpiralAlgorithm.java – interfejs definiujący wymagane metody w algorytmach
c) exceptions – pakiet zawierający wyjątki potrzebne do obsługi pewnych sytuacji wyjątkowych:
-WrongArgumentException.java – wyjątek rzucany, gdy któryś z parametrów wywołania jest niepoprawny
d) model – pakiet zawierający pliki z jednostkami, na których operujemy:
-Point.java – klasa określająca punkt na płaszczyźnie
-Vector.java – klasa określająca wektor między dwoma punktami
e) utilities – pakiet klas „narzędzi" dla programu
-AngleComparator.java – klasa określająca sposób porównywania dwóch punktów na podstawie kątów
-ArgumentParser.java – klasa zajmująca się przetworzeniem parametrów wejściowych aplikacji i odpowiedniego wywołania funkcji
-Measurement.java – klasa służąca do przeprowadzania pomiarów i testowania złożoności czasowej
-ModulusComparator.java – klasa określająca sposób porównywania dwóch punktów na podstawie ich odległości od punktu zadanego
-PointGenerator.java – klasa odpowiedzialna za generowanie losowych punktów na płaszczyźni
f) view – pakiet zawierający klasy określające widok aplikacji
-View.java – klasa przechowująca wymiary okna oraz główny *Stage* programu
-SpiralView.java – klasa określająca sposób wyświetlania i rysowania spirali

6. Testy złożoności oraz pomiary czasu wykonania

Do wykonania testów złożoności wykorzystano tryb -m3 uruchomienia programu. Dla każdego z algorytmów wykonano 11 testów. Początkowe dane wejściowe miały rozmiar 10000 i zwiększały się o 2000 co test, dając największy rozmiar o wielkości 30000. Ilość iteracji do uśrednienia wyniku dla każdego testu wyniosła 10.

javac AAL/AAL.java
java AAL.AAL -m3 -n10000 -s2000 -t11 -i10

Algorytm rosnących promieni – spodziewana złożoność *O(n * log(n))*		
n	**t (ms)**	**q(n)**
10000	30.5	0.624
12000	50.7	0.847
14000	65.6	0.924
16000	61.7	0.750
18000	76.3	0.815
20000	110.9	1.000
22000	126.9	1.087
24000	116.2	0.904
26000	125.0	0.891
28000	147.6	0.970
30000	161.0	0.981

Dodatkowy test dla algorytmu rosnących promieni (od 30 tysięcy, co 5 tysięcy, 7 testów, po 10 iteracji):

Algorytm rosnących promieni – spodziewana złożoność *O(n * log(n))*		
n	**t (ms)**	**q(n)**
30000	149.7	0.650
35000	232.0	0.851
40000	275.5	0.873
45000	380.6	1.000
50000	361.6	0.898
55000	448.2	1.003
60000	482.9	0.983

Algorytm otoczek wypukłych – spodziewana złożoność *O(n^2 * log(n))*		
n	**t (ms)**	**q(n)**
10000	2375.7	1.220
12000	3254.9	1.138
14000	4249.2	1.074
16000	5551.4	1.059
18000	6881.5	1.025
20000	9277.6	1.000
22000	10087.1	0.986
24000	12012.8	0.978
26000	14681.6	1.010
28000	18610.2	1.096
30000	18537.5	0.945

Wartości q w powyższych tabelach oscylują w okolicach wartości 1. Oznacza to, że podane złożoności obliczeniowe są zgodne z rzeczywistością. Widoczne są jednak pewne skoki w algorytmie rosnących promieni. Może to być spowodowane wpływem czynnika liniowego na czas wykonywania obliczeń. Wpływ tego czynnika powinien zanikać wraz ze wzrostem ilości danych wejściowych n. W algorytmie otoczek wypukłych może być niewidoczny wpływ czynnika log(n), jednak on także wraz ze wzrostem danych będzie wzrastał.

7. Podsumowanie

Przeanalizowane algorytmy zostały sprawdzone pod kątem ich działania, poprawności i szybkości. Wszystkie obliczenia dotyczące złożoności obliczeniowej znalazły potwierdzenie w wartościach zmiennej q, choć dużo lepsze wyniki uzyskuje się dla większych wartości n, jednak znacznie wydłuża się wtedy proces wyliczania. Zaimplementowane metody pozwalają na w miarę szybkie i dokładne wygenerowanie spirali ze zbioru danych punktów.