Tomasz Murawski, 25.01.2018

Analiza Algorytmów
Projekt PW.7
Dokumentacja końcowa

Zadanie

Na zadanym prostokącie znajduje się *n* punktów. Należy wskazać na tym prostokącie największe koło, w którym nie ma żadnego punktu.

Przyjęte założenia

- $n \geq 1$
- Współrzędne punktów są liczbami rzeczywistymi
- Boki prostokąta są równoległe do osi układu współrzędnych
- Punkty mogą leżeć na bokach prostokąta
- W razie istnienia więcej niż jednego największego pustego koła, program zwraca dowolne z nich

Opis problemu

Dane zadanie jest przykładem problemu największego pustego koła (Largest Empty Circle, LAC). Okrąg największego pustego koła w danym prostokącie będzie zawsze przebiegał przez co najmniej trzy punkty – mogą to być zarówno punkty ze zbioru n danych punktów, jak i punkty należące do boków prostokąta będących w takich przypadkach fragmentami stycznych do okręgu. Stąd wyróżniłem pięć możliwych kombinacji, w jakich znajdować się może okrąg szukanego największego koła:

1. Przechodzi przez 3 z *n* punktów
2. Przechodzi przez 2 z *n* punktów i jest styczny do jednego boku prostokąta
3. Przechodzi przez 1 z *n* punktów i jest styczny do dwóch sąsiednich boków prostokąta
4. Przechodzi przez 1 z *n* punktów i jest styczny do dwóch przeciwległych boków prostokąta
5. Nie przechodzi przez żaden z *n* punktów i jest styczny do trzech boków prostokąta (największe możliwe koło zawierające się w prostokącie)

Proponowane rozwiązania

Każdą z wymienionych wyżej sytuacji należy osobno rozpatrzyć. W przypadku (1.) najprostszym, ale i najbardziej kosztownym rozwiązaniem jest sprawdzanie wszystkich możliwych kombinacji trzech punktów, obliczanie leżącego na nich okręgu oraz sprawdzenie czy nie leży on poza prostokątem oraz czy nie zawiera innych punktów. Można jednak zauważyć, że środki rozpatrywanych okręgów będą się pokrywały z wierzchołkami diagramu Woronoja, które to można obliczyć algorytmem Fortune'a przy znacznie korzystniejszej złożoności obliczeniowej $O(n \log n)$.

W sytuacjach opisanych w punktach (2.) – (4.) środki okręgów najczęściej nie będą pokrywały się z wierzchołkami diagramu Woronoja, toteż niezbędne będzie użycie algorytmów „brutalnych" rozpatrujących każdą możliwą kombinację wierzchołków i boków, wyliczanie okręgów oraz sprawdzanie czy w ich wnętrzu nie znajdują się pozostałe punkty.

W przypadku (5.) wystarczy rozpatrzeć dwa przypadki (okrąg przylegający do jednego z dwóch krótszych boków prostokąta) i dla każdego z nich przeszukać wszystkie wierzchołki, sprawdzając czy leżą one wewnątrz danego koła. Ponieważ przypadek (5.) opisuje największe możliwe koło wpisane w prostokąt, warto rozważyć go w pierwszej kolejności, gdyż w przypadku gdy takie koło faktycznie istnieje, da się je wykryć przy złożoności O(n).

Rozwiązanie problemu

W celu wykrycia okręgów należących do grupy przypadków (1.) użyłem nieco zmodyfikowanego algorytmu Fortune'a do generacji diagramu Woronoja i tym samym wykrywania środków owych okręgów i obliczania ich promieni.

Algorytm Fortune'a polega na operowaniu dwoma strukturami: linią brzegową (beach line) oraz miotłą (sweep line). Miotła to w praktyce linia prosta, przesuwająca się po badanej płaszczyźnie w stałym kierunku, natomiast linia brzegowa to krzywa powstała z połączenia sąsiednich parabol, których ogniskami są mijane przez miotłę punkty, a kierownicą sama miotła. Sąsiadujące parabole tworzą między sobą odcinki będące krawędziami grafu Woronoja, rozpinające się pomiędzy wierzchołkami grafu Woronoja, czyli punktami, w których spotkały się trzy sąsiadujące parabole. Napotkanie przez miotłę punktu nazywa się zdarzeniem wierzchołkowym (site event), natomiast spotkanie się trzech parabol to zdarzenie okręgu (circle event).

Różnica między moją implementacją algorytmu Fortune'a a oryginałem polega na użyciu innej struktury danych do przechowywania w pamięci linii brzegowej. W pierwotnej wersji algorytmu jest ona przechowywana jako drzewo binarne, którego węzłami są punkty będące ogniskami paraboli (w przypadku liści drzewa) oraz krawędzie diagramu Woronoja (w przypadku innych węzłów). W moim programie zaś postanowiłem zastosować w tym celu listę dwukierunkową, mając w uwadze wygodę operacji dokonywanych na danym kontenerze, ale także brak widocznej z mojej perspektywy znaczącej przewagi ze strony drzewa.

Dla kontrastu drugą metodą poszukiwania okręgów w wierzchołkach diagramu Woronoja jest standardowy algorytm typu brute-force, badający wszystkie możliwe kombinacje trójek wierzchołków. Dla każdej takiej trójki sprawdza on czy okrąg oparty na tych punktach nie zawiera żadnego innego punktu.

Złożoność

Na całkowitą złożoność mojego rozwiązania składają się algorytm Fortune'a dla przypadków (1.) oraz algorytmy brute-force dla pozostałych przypadków. Z racji braku alternatywnego rozwiązania dla przypadków (2.) - (5.) postanowiłem skupić się przede wszystkim na przypadku (1.) i dokładnie zbadać złożoność samego algorytmu generacji diagramu Woronoja.

Mimo że w oryginalnym algorytmie Fortune'a teoretyczna złożoność wynosi *O(n log n)*, to z racji zamiany struktury drzewa na dwukierunkową listę złożoność pesymistyczna w moim algorytmie wzrosła do $O(n^2)$, z racji n wydarzeń do obsłużenia oraz n operacji potrzebnych na jej obsłużenie (przeszukiwanie listy). Dodatkowo, przed każdym rozpoczęciem algorytmu sortuję tablicę punktów wedle malejących wartości współrzędnej y, aby miotła przechodząca po płaszczyźnie od góry do dołu mogła po kolei napotkać kolejne punkty, co stwarza z tablicy punktów swojego rodzaju kolejkę zdarzeń wierzchołkowych. Punkty sortuję funkcją *sort* z biblioteki <*algorithm*>. Koszt sortowania wynosi *O(n log n)*, zatem nie jest czynnikiem dominującym w kwadratowej złożoności mojego algorytmu.

Pesymistyczna teoretyczna liczba potrzebnych do wykonania operacji wynosi zatem $n^2 + n \log n$. W poniższej tabeli przedstawione są wyniki pomiarów czasu wykonania algorytmu Fortune'a dla różnych liczb danych punktów.

N	t(n) [s]	T(n)	q(n)
1000	0,0026708	1009966	12,86091
5000	0,0172154	25061439	3,340832
10000	0,0441666	100132877	2,145165
20000	0,115094	400285754	1,398383
35000	0,251988	1225528327	1
50000	0,419498	2500780482	0,815827
75000	0,75268	5626214595	0,650635
90000	0,96633	8101481187	0,580102
100000	1,13909	10001660964	0,553898

Gdzie:

N – liczba danych punktów
t(n) – pomiar czasu wykonania operacji (w sekundach)
T(n) – szacowana liczba wykonanych operacji
q(n) – stosunek $\frac{t(n)}{c*T(n)}$, gdzie c = $\frac{t(n_{mediana})}{T(n_{mediana})}$

Dla porównania algorytmu Fortune'a z algorytmem brutalnym zmierzyłem również czasy wykonania programu dla tego samego zestawu punktów dla obu algorytmów. Wyniki widać w poniższej tabeli:

N	Fortune [s]	Brute-force [s]
150	0,000429992	0,0829535
200	0,000341593	0,242432
250	0,000423896	0,504522
300	0,00050832	0,974485
350	0,000626614	1,71046
400	0,000726921	2,83894
450	0,00084458	4,48325
500	0,00096376	6,51327
550	0,00107389	9,37034
600	0,00118617	13,7138
650	0,00134751	18,741
700	0,00149275	23,4293
750	0,00152896	30,7872
800	0,00166123	39,4979
850	0,00184507	52,9972

Wnioski

Algorytm generacji grafu Woronoja metodą Fortune'a okazał się nieporównywalnie skuteczniejszy od algorytmu brutalnego, co widać na powyższej tabeli. Już dla zaledwie 150 elementów pierwszy algorytm wyprzedzał drugiego w czasie wykonania o dwa rzędy wielkości, natomiast po zwiększeniu

liczby elementów o 700, w szybkości algorytmu Fortune'a zaszła zmiana o jeden rząd wielkości, podczas gdy algorytm brutalny zbliżał się już do granicy jednej minuty.

Sam algorytm Fortune'a, mimo że ze zmienioną strukturą linii brzegowej, okazał się skuteczniejszy niż moje teoretyczne szacunki. Świadczy o tym parametr *q(n)*, którego zmniejszanie wraz z rosnącym N oznacza, że złożoność obliczeniowa została przeszacowana i w praktyce jest lepsza niż $O(n^2)$.