Line Sweep Algoritmaları

Sweep line yöntemi, bir doğruyu ilerletirken bu doğrunun etrafında bazı hesaplamalar yaparak problem çözme yöntemidir. Tabi bütün noktaları aynı anda işleme almayacağız, noktaları doğrumuza göre ayrı olarak işleme alacağız.

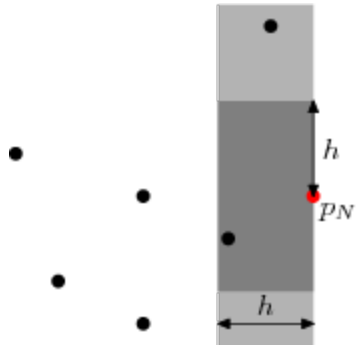
Bir çok yerde referansımız Manhattan ve Euclid uzaklık sistemleri olacak. Euclid uzaklığı normal olarak Pisagor teoremiyle hesapladığımız uzaklık. Manhattan uzaklığı ise (x1,y1) ve (x2,y2) noktalarının koordinat eksenlerine paralel gidilerek hesaplanan uzaklığı($|x1 - x2| + |y1 - y2|$ şeklinde hesaplanır)(taxicab uzaklığı da denir).

Ek olarak bazı algoritmalarda dengeli ikili ağaçlarara(balanced binary trees) ihtiyacımız olacak. Bu konuda genel olarak C++'daki set ve Java'daki TreeSet işimizi görecek lakin bazı durumlarda fazladan bilgi depolamamız gerektiğinden bu hazır yapılar yeterli olmayacak.

Closest Pair(En Yakın İkili)

Bu problemde verilen bir noktalar kümesindeki bütün nokta ikililerini düşündüğümüzde en yakın olan ikiliyi bulmamız isteniyor(iki uzaklık sistemine göre de). Akla ilk gelen çözüm olabilecek bütün ikilileri elden geçirmek ki bu çözüm zaman karmaşıklığı $O(N^2)$. Ama bir sweep line algoritmasılaya bu problemi O(NlogN) zaman karmaşıklığında çözebiliriz.

Doğrumuzun 1'den N-1'e kadar her noktayı taradığını düşünün, yani doğrumuz şu an N. noktada(noktalar x koordinatlarına göre küçükten büyüğe doğru sıralı). Şu ana kadar bulduğumuz en küçük uzaklığa H diyelim. Şu an yapmamız gereken N. noktaya H mesafesinden daha yakın bir nokta bulmak. N ile x koordinatları farkı H'den küçük eşit olan ve doğrumuzun daha önce üzerinden geçtiği noktaların kümesini tutuyoruz(şekilde açık gri ile gösterilen bölge). Doğrumuz bir noktayı geçtiğinde o nokta kümeye eklenir. Doğrumuz yeni bir noktaya geçtiğinde ve H azaldığında gereksiz noktalar kümeden çıkartılır. Kümenin içindeki sıralama y koordinaına göredir. Dengeli ikili ağaç bu işlemler için uygundur(logN'lik kısım için).

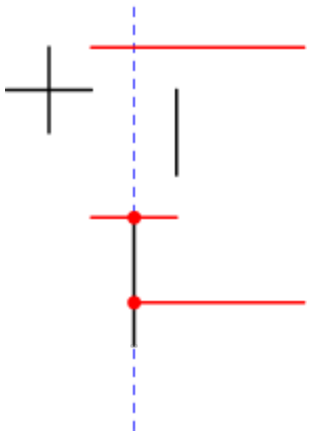


N. noktaya H'den daha yakın olan noktaları bulmak için sadece aktif olan kümemizdeki noktalara bakmamız yeterlidir(sol tarafta o kümede olmayan herhangi bir noktanın sadece x koordinatı bile N. noktanın x koordinatına H'den daha uzaktır). Ayrıca aynı sebepten ötürü kümenin içindeki noktalardan y koordinatı yN-H'den küçük olanlara ve yN+H'den büyük olanlara da bakmamıza gerek yok(yN, N.

noktanın y koordinatıdır)(bu kısıtı da ekleyince kümemiz koyu gri olan kısım oldu). Bu aralığı O(logN) işlemde bulabiliriz, daha da önemli olan şey ise bu aralıkta en fazla O(1)(asıl sayı kullanılan metrik sisteme bağlıdır) eleman olabileceğidir. Çünkü bu kümedeki herhangi iki nokta birbirine H'den daha yakın değildir. Buradan anlaşılacağı üzere her nokta için sorgumuz O(logN) işlem tutacak, algoritmanın toplamdaki çalışma zamanı ise O(NlogN) olacaktır.

Line Segment Intersections(Doğru Parçası Kesişimleri)

Sadece koordinat eksenlerine paralel doğru parçalarının kesişimlerinin sorulduğu problem ile başlayacağız. Yatay olan doğru parçalırının birden fazla x koordinatı olduğu için elemanları doğrudan x koordinatlarına göre sıralamayacağız. Bunun yerine x koordinatlarını durumlarına göre düşüneceğiz. Bir x koodinatında 3 farklı durum olabilir: yatay bir doğru parçasının başlangıcı, yatay bir doğru parçasının bitişi ve dikey bir doğru. Sweep linei uygularken kaydırdığımız doğrunun o an kestiği yatay doğru parçalarının y koordinatlarına göre sıralanmış hâllerinin kümesini tutacağız(şekildeki kurmızı doğru parçaları).



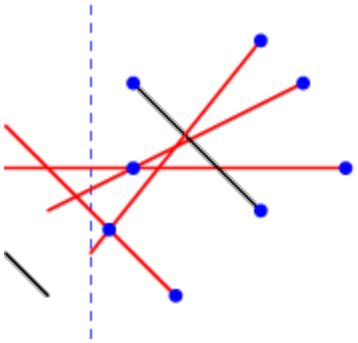
Elimize yatay bir doğru parçasının başı geldiğinde onu kümeye ekliyoruz, sonu geldiğinde ise kümeden çıkarıyoruz. Hatırlatalım, küme ile ilgili eleman ekleme-çıkarma işlemleri için dengeli ikili ağaçları kullanabiliriz. Bu apaçlar sayesinde her işlemi O(logN)'de yapmış oluruz. Dik doğru parçası durumlarında ise kümemize hızlı bir arama algoritması uygulamalıyız. Eğer doğrular birbirlerini kapsayabiliyorsa ek olarak bazı işlemler yapmamız gerekir, ayrıca sadece doğru parçalarının birinin ucunun diğerine değmesi durumunun da kesişim sayılıp sayılmadığı önemli(probleme göre değişir). Tabi bu durumlar zaman karmaşıklığımızı etkilemeyecek.

Kesişim noktalarının sadece sayısın değil koordnatlarını da bulmak istiyorsak O(NlogN+I) tane işlemi gözden çıkarmamız gerekir(I, kesişim noktalarının sayısı). İkili ağacımızı geliştirerek(özellikle her nodeun alt ağacında kaç tane node olduğunu tutmasını saplayarak) kesişimleri O(NlogN) işlemde sayabiliriz.

Daha genel düşündüğümüzde aslında doğru parçalarının dik yatay olmalarına gerek olmadığını görürüz. Yani aktif kümedeki elemanlar kesişim durumunda yer değiştirebilirler. Her durumu önceden

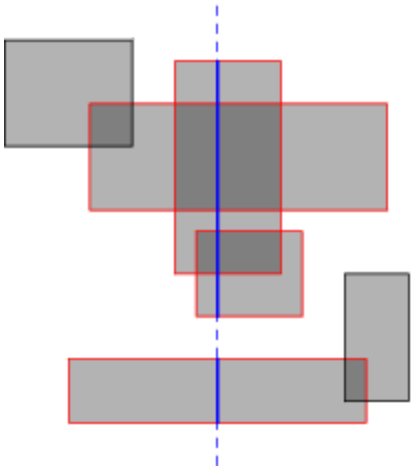
sıralanmış bir şekilde tutmak yerine kümeye kesişim durumlarını dinamik olarak eklemek ve kümeden çıkarmak için priority queue kullanacağız. Priority queueu(PQ) doğru parçalarının içerecek.

Herhangi bir anda PQ, doğru parçalarının bitiş notalarının durumlarını içerir(aktif kümedeki ardışık kesişimleri de içerir). Toplamda O(N+I) durum olmuş olacağına ve her durum O(logN) işlem gerektireceğine göre algoritmanın zaman karmaşıklığı O(NlogN+IlogN) olur. Şekilde ileride gelecek olan durumlar mavi ile gösterilmiştir. İlerideki bütün kesişimlerin sırada olmadığına dikkat edin. Çünkü ya o doğrular henüz aktif değildir ya da ikisi ardışık değildir.



Bir Düzlemde Dikdörtgenlerin Kapladığı Alan

Bir düzlemde, kenarları koordinat eksenlerine paralel olarak verilen dikdörtgenlerin ne kadarlık bir alanı kapladığını hesaplayacağız. Doğru parçalarının kesişimleri problemi gibi bu problemi de durumlar ve aktif kümenin yardımıyla çözebiliriz. Her dikdörtgenin iki durumu var: sol kenar ve sağ kenar. Sol kenarı geçtiğimizde dikdörtgeni aktif kümeye ekleyeceğiz sağ kenarı geçtiğimizde ise dikdörtgeni aktif kümeden çıkaracağız.



Şu anki durumda doğrumuzun hangi dikdörtgenleri kestiğini biliyoruz(şekilde kırmızı kenarlı olanlar). Ama bize lazım olan o an doğrumuzla kesişen kısımların toplam uzunluğu(şekildeki mavi kısım). Bu uzunlukla o an arasında bulunduğuuz iki durumun uzaklığını çarptığımızda o aradaki alanı hesaplamış oluruz.

Mavi kısmın uzunluğunu başka bir döngüyle hesaplayabiliriz(bu döngü yuarıdan aşağıya doğru gidecek). Aktif olmayan dikdörtgenleri göz ardı edip yukarıdan aşağıya bir line sweep algoritması çalıştırdığımızı düşünün. Bu aşamada durumlarımız yatay kenarlar, bunları teker teker ele alacağız ve o an kaç tane yaty doğrunun aktif olduğunu bir sayçta tutacağız, bu şekilde mavi çizginin uzunluğunu hesaplayabiliriz. Tabi sürekli ekleme ve çıkarma yapmayacağız durumdan duruma geçerken yapacağız.

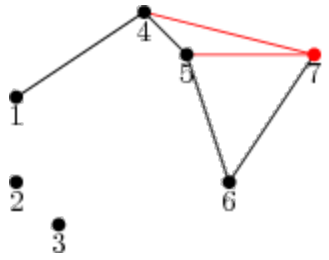
Doğru veri yapısıyla bunu $O(N^2)$'de çalıştırabiliriz(ipucu: aktif küme için ağaç yerine 1-0 dizisi kullanın ve yatay kenarları kümesini önceden sıralayın). Aslında yukarıdan aşağıya doğru işlediğimiz line sweep algoritmasını dengeli ikili ağaçlar(balanced binary trees) yardımıyla geliştirebiliriz ve çalışma zamanını O(NlogN)'e getirebiliriz, lakin bu bir geometri probleminden çok veri yapıları problemi olmuş oldu. Bu tarz algoritmalar bir düzlemdeki diktörtgenlerin çevresi ve en çok dikdörtgenin içinde olan noktayı bulma gibi problemlerde kullanılabilir.

Convex Hull

Verilen noktalar kümesini çevreleyen en küçük konveks(dışbükey) çokgeni bulma problemini inceleyeceğiz. Convex hullı bulmanın bir çok yöntemi vardır. En iyilerinden biri açılara göre bir sıralama gerektiren Graham Scan yöntemidir. İlk başta göründüğü kadar kolay değildir, çünkü açıları hesaplamak maliyetli bir işlem ve sayısal hatalar çok olası. Daha basi ve aynı derecede verimli başka bir algoritma ise Andrew'inkidir ve gereken tek sıralamayı line sweep için x koordinatları üzerinedir(normalde Andrew'ınki y koordinatlarına göre sıralayıp bir kaç geliştirme kullanır ama onlar burada anlatılmamıştır).

Andrew'un algoritması convex hullı üst ve alt olmak üzere iki parçaya ayırır. Genellikle bu iki parça uçlardan birleşirler, fakat minimum(ya da maksimum) x oordinatına sahip birden fazla nokta varsa bunlar dik bir doğru parçasıyla birleştirilirler. Sadece üst kısmın naıl oluşturulacağını anlatacağız, alt kısım da benzer şekilde oluşturulabilir hatta aynı döngü içinde bile yapılabilir.

Üst kısmı oluşturmaya x koordinatı en küçük olan noktadan başlayacağız(eşitlik durumunda en büyük y koordnatına sahip olandan başlayacağız). Sonra noktaları x koordinatlarına göre küçükten büyüğe doğru gezerek ekleyeceğiz(eşitlik durumlarında her zaman y koordinatı büyük olanı alacağız). Tabi bazen dışbükeylik durumu bozulabilir:



Siyah çizgiler hullın şu anki hâli. 7 numaları noktayı ekledikten sonra son üçgenin(5,6,7) dışbükey oup olmadığını kontrol ediyoruz. Dışbükey olmadığı durumda sondan ikinci noktayı siliyoruz(bu durumda 6 numaralı nokta). BU ügen kontrol ve silme işleini son üçgen dışbükeyliği bozmayana kadar sürdürüyoruz(son üçgenimiz bu durumda 1,4,7 olacak). Bu temelde Graham Scan'in algoritmasıyla aynı şey fakat biz noktaları bir referans noktasıyla yaptıkları açıya göre değil, x koordinatlarına göre işlemealıyoruz. İlk başta bu işlemlerin zaman karmaşıklığı $O(N^2)$ gibi görünse de

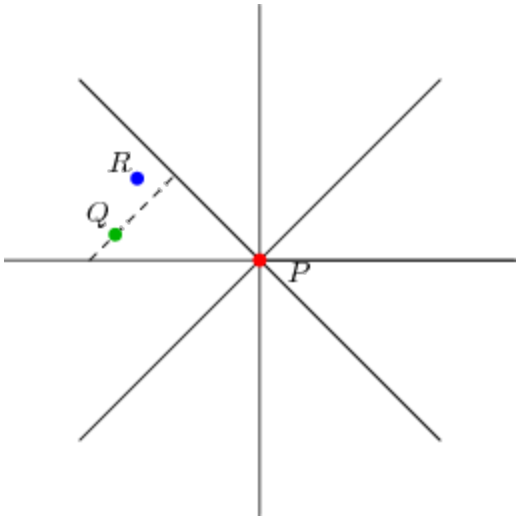
aslında O(N)'dir. Çünkü her nokta hulla bir kez girebilir ve hulldan bir kez silinebilir. İlk baştaki sıralam işleminden doayı algoritmanın toplamda zaman karmaşıklığı O(NlogN)'dir.

Manhattan Minimum Spanning Tree

Divide and conquer algoritmalarıyla line sweep algoritmalarını birleştirerek daha güçlü algoritmalar elde edebiliriz. Mesela verilen bir noktalar kümesinin MST'sini bulmak(noktalar arasındaki uzaklığı manhattan uzaklığı olarak düşündüğümüz durumda). Aslında bu algoritma Guibas ve Stolfi tarafından anlatılmıştır.

Önelikle bunu daha basit bir probleme indirgeyelim. Normal MST algoritmaları(örn: Prim'in algoritması) MST'yi O((E+N)logN) zaman karmaşıklığında hesaplayabilir(E, edge sayısıdır). Eğer bazı geometrik özellikleri kullanarak edge sayısını O(N)'e indirebilirsek algoritmanın çalışma zamanı O(NlogN) olur.

Her nokta için düzelmi 8'e bödüğümüzde sadece her bölmedeki en yakın 8 komşusunu göz önünde bulundurmamız yeterli(ayrıntılar için aşağıdaki şekli inceleyiniz). Şekilde durum bir bölümde açıklanıyor. P noktasının o blümdeki en yakın komşusu Q'dur(çizgili doğru üzerindeki bütün noktalar P'ye eşit uzaklıktadır). R ise o bölümdeki başka bir nokta. Eğer PR yapımızdaki bir edge ise daha iyi bir yapı için onun yerine PQ veya QR kullanılabilir. Çünkü bölmenin şekli $|QR| \leq |PR|$ eşitsizliğinin doğruluğunu garanti ediyor. Böylece yapıyı kurarken PR'yi hesaba katmamız gerek kalmıyor.



Bu şekilde problemi her nokta için 8 bölgedeki en yakın komşusunu bula problemine indirgemiş olduk. Sadece şekilde gösterilen bölmeye göre olan çözümü anlatacağız diğer bölmeler de benzer yöntemlerle çözülebilir. Şekildeki bölmenin çözümü en büyük x-y'ye sahip olan noktayı bulmaktır, x+y'lr üzerine upper bound ve y'ler üzerine lower bound yardımıyla bulunabilir, bu şekilde problemi istediğimiz biçime indirgemiş olduk.

Şimdi y'ler üzerine lower bound olmadığını düşünün. Bu durumda problemi her P noktası için rahatça çözebiliriz. Noktaları x+y'lerine göre artan sırada gezeceğiz(line sweep) ve Q, o ana kadarki en büyük x-y'ye sahip olan nokta olacak. İşte bu aşamada divide and conquer devreye girecek. Noktalar kümesini yatay bir doğruyla ikiye böleceğiz, ve böldüğümüz her kısım için problemi özyinelemeli(recursive) olarak çözceğiz. Üst kısımdaki P'ler için fazladan bir şey yapmaya gerek yok, çünkü aşağı kısımdaki noktalar herhangi bir P'nin Q'su olamazlar. Alt kısmı çözerken üst kısma

bakmazsak gerekli noktayı kaçırabiliriz. bu nokataları da öncekine benzer bir şekilde dikkate almalıyız: noktaları x+y'lerine göre artan sırada gezip o ana kadarki en iyi noktayı(x-y'si en büyük olan) tutmalıyız ve aşağı kısımdaki her nokta için üst kısımdaki en iyi noktanın buradaki en iyi noktadan daha iyi olup olmadığını kontrol etmeliyiz.

Şimdiye kadar her noktalar kümesinin y'ye göre düzgün bir biçimde bölünebileceğini kabul ettik ve x+y'lerine göre noktaları nasıl gezeceğimize açıklık getirmedik. Aslında bu yöntemin en güzel yönlerinden biri merge sort ile temelde aynı olmasıdır. Her kısmı x+y'lerine göre sıralamış olduğumuz için bunları merge-sort mantığıyla kullanabiliriz(başta hepsi y'lerine göre sıralı). Bu şekilde çalışma zamanı O(NlogN) olur.

Belirli bir açı içindeki en yakın noktayı bulma algoritması, MST probleminin Euclid uzaklığı versiyonunda da kullanılabilir, fakat O(NlogN) en kötü durumlar için garanti olmaz çünkü uzaklıkları dorusal bir eşitliğe bağlayamayız. Yine de Euclidian MST'yi O(NlogN)'de hesaplayabiliriz(Delaunay triangulation).

Örnek Problemler

Box Union(http://community.topcoder.com/stat?c=problem_statement&pm=4463&rd=6536)
Culture Growth(http://community.topcoder.com/stat?c=problem_statement&pm=3996&rd=7224)
Power Supply(http://community.topcoder.com/stat?c=problem_statement&pm=5969)
Convex Polygons(http://community.topcoder.com/stat?c=problem_statement&pm=4559&rd=7225)