Youtube Video Linki

https://youtu.be/A8h6KFmikb8

Mert Arıcan
M. Fatih Amasyalı
18 Haziran 2020

Shamos'un Algoritması (Dönen Kumpaslar Metodu)

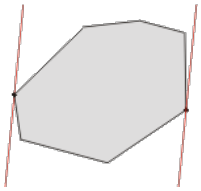
Michael Ian Shamos'un, bir konveks çokgenin çapını bulmaya yarayan algoritması

Shamos'un algoritması, ilk kez Michael Ian Shamos tarafından 1978'de yayınlanan "Computational Geometry" adlı tezinde—ki daha sonra bu tez, bilgisayar bilimlerinde "Computational Geometry" alanının başlangıcı kabul edilecektir—kullanılmıştır. Shamos, bu algoritmayı, standart formda verilmiş bir konveks çokgenin çapını (bir konveks çokgenin çapı en uzun köşegenine eşittir) bulmak için kullanmıştır. Daha sonra, Goudfried Toussaint tarafından yapılan çalışmada ise algoritmanın çok daha farklı geometrik problemler için genelleştirilip kullanılabileceği ve çoğu kullanımında da en uygun (optimal) çalışma süresi vereceği gösterilmiştir. Ayrıca Toussaint, metoda "Dönen Kumpaslar" (Rotating Calipers) adını vermiştir. Buradaki kumpas "hile, düzen" manasında değil "sanayide kalınlık ve incelikleri ölçmede kullanılan ölçüm aleti" anlamındadır. Metodun çalışması, iki tane kumpasın, konveks çokgenin etrafında dönmesine ve her dönüşte de kumpasın çokgenin kenarını kavrayacak kalınlık ve inceliğe ayarlanmasına benzetildiği için (…rotating a pair of dynamically adjustable calipers once around the polygon. [2]) bu ad verilmiştir. Daha sonra Toussaint'in doktora öğrencisi Hormoz Pirzadeh tarafından yayınlanan doktora tezinde bu genelleştirmelerin sayısı artırılmıştır ve Toussaint'in "Dönen Kumpaslar" ile ilgili eksik bıraktığı kanıtlar da eklenmiştir. Algoritma, "Computational Geometry" alanında, birbirinden farklı yapıda problemleri çözmeye yarayan genel bir metot, yaklaşım veya bir paradigma olarak kabul edilmiştir.

Problemin Tanımı

Shamos tarafından çözümü sunulan orijinal problem, köşe noktaları verilen bir konveks çokgenin çapını (en uzun köşegenini) bulmaktır. Çözüm bazı geometrik temellere dayanır,

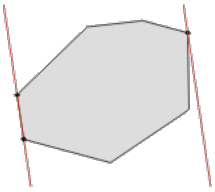
örneğin çokgenin içindeki bir doğru parçasının çap olabilmesi için, bir antipodal nokta çiftinden geçiyor olması gerekir. Antipodal nokta çifti şöyle tanımlanabilir: Eğer çokgenin köşe noktalarından olan iki p ve q noktasından birbirine paralel şekilde geçen destek doğruları var ise bu iki noktaya antipodal nokta çifti denir. Destek doğrusu ise şu şekilde tanımlanabilir: bir konveks çokgen verilmiş olsun, eğer doğru çokgenle kesişiyorsa ve çokgenin iç kısmı doğrunun tek bir tarafında kalıyorsa bu doğruya destek doğrusu denir. Görselleştirmek gerekirse:



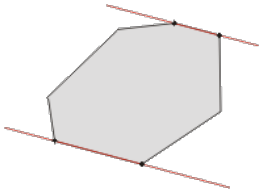
Bu iki doğrunun her biri bir destek doğrusudur, çünkü bu doğrular çokgenle kesişir ve çokgen bu doğruların tek bir tarafında bulunur. Ek olarak bu iki doğru birbirine paralel olduğunda bu noktalar bir antipodal çifttir.

figür 1.1

Ayrıca aşağıdaki görsellerde destek doğrularını ve antipodal nokta çiftlerini gösterir:



figür 1.2



figür 1.3

Soldaki şekilde iki, sağdakinde ise dört adet antipodal nokta çifti vardır.

Daha önceden, bir doğru parçasının, konvek bir çokgenin çapı olabilmesi için antipodal nokta çiftinden geçmesi gerektiğinden bahsedilmişti. Şimdi daha kesin bir tarif verilecek olursa, bir konveks çokgenin çapı, antipodal nokta çiftlerinden geçen doğru parçalarından uzunluğu en büyük olanıdır denebilir.

Basit Yaklaşım

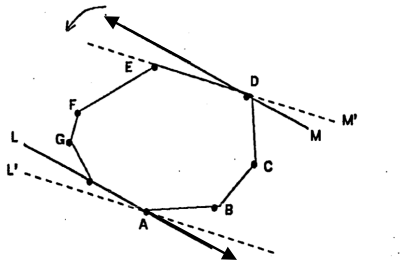
Problemin çözümüne yönelik en basit yaklaşım, köşe noktalarını dikkate alarak tüm nokta ikililerinin oluşturulması, bu nokta ikililerinden geçen doğru parçalarının oluşturulması ve aralarından uzunluğu en fazla olanın seçilmesidir. Bir çokgendeki her bir nokta çiftinin oluşturulması $O(n^2)$ algoritmik karmaşıklığa sahiptir. Bu yaklaşım doğru cevabı verse de algoritmik karmaşıklık açısından maliyetlidir.

Shamos'un Algoritması

Daha önceden, bir konveks çokgenin çapının, antipodal nokta çiftleri arasında çizilen doğru parçalarından en uzun olanı olduğundan bahsedilmişti. Shamos'un algoritması bu geometrik temele dayanır, çokgendeki tüm antipodal çiftler tespit edilir ve daha sonra basit bir karşılaştırmayla uzunluğu en büyük olan çift belirlenir.

M. Shamos'un, tüm antipodal nokta çiftlerini bulmak için sunduğu algoritma şöyle tarif edilebilir:

-Başlangıç için bir antipodal nokta çifti belirlenir. (En büyük ve en küçük y değerlerine sahip köşe noktaları başlangıç antipodal noktaları olarak kullanılabilir. Shamos farklı bir yöntem sunar.)



Burada ilk antipodal nokta çifti A ve D dir. Bu noktalardan M ve L destek doğruları geçer. Bunlar algoritmadaki 'kumpas'lara karşılık gelirler.

- M sol, L ise sağ yöne döndürülecektir (aslında saat yönünün tersinde, birbirini takip edecek şekilde dönecekler), bu döndürme ise, L'nin, sağında bulunan kenar (yukarıdaki örnekte AB kenarı) ile arasındaki açı, ve M'nin de solunda bulunan kenar (yukarıdaki örnekte DE kenarı) ile arasındaki açının büyüklüğüne göre yapılacaktır. Hangisinin açısı küçükse (yukarıdaki örnekte M'in), her iki doğru da kendi yönünde, küçük olan açı farkı kadar döndürülecektir. Kumpasların döndürmeden sonraki durumu M' ve L' ile gösterilmiştir. Bu durumda küçük açı farkı olan kumpas, (M), yeni bir noktaya, (E), değer. Bu durumda E ve A bir antipodal nokta çifti olarak eklenir. Bir sonraki döndürme için M' ile EF kenarı arasındaki açı ve L' ile AB arasındaki açı karşılaştırılır. Daha sonra bu süreç çokgen etrafında bir tam tur tamamlanana kadar devam eder. Eğer birbirine paralel kenarlar denk gelirse 3 antipodal nokta eklenir. Tam tur tamamlandığında çokgendeki tüm antipodal nokta çiftleri belirlenmiş olur. Bu çiftlerden geçen doğru parçalarının en uzun olanı çokgenin çapıdır.

Algoritmanın Teorik Olarak İncelenmesi ve Karmaşıklık Analizi

$P = \{p_1, p_2, \ldots, p_n\}$ standart formda, yani köşeleri kartezyen koordinat sisteminde saat yönünde sıralanmış şekilde ve ardışık üç köşe noktası birbirine paralel olmayacak şekilde verilmiş bir konveks çokgen olsun.

Tanım 2.1.1 Bir L doğrusu, konveks bir çokgenle kesişiyorsa ve çokgenin iç kısmı doğrunun tek bir tarafında kalıyorsa bu doğru bir destek doğrusudur.

Tanım 2.1.2 Bir P konveks çokgeni verilmiş olsun; p, q ∈ P olan iki noktadan, birbirine paralel iki destek doğrusu geçiyor ise bu noktalara antipodal çift denir.

Teorem 2.1.1 P konveks çokgeninin çapı, P'nin birbirine paralel olan destek doğruları arasındaki en büyük uzaklıktır.

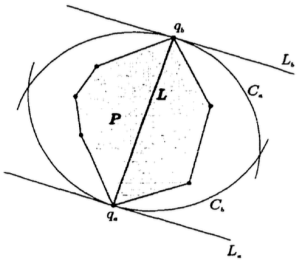
Kanıt: P = {p_1,p_2,…,p_n} verilmiş olsun. {q_a,q_b} köşe çiftinin P'nin çapını belirlediğini varsayalım. d_{ab} = uzaklık(q_a, q_b) olsun, ve C_a ise q_a merkezli d_{ab} yarıçapında bir çember olsun, ve C_b ise q_b merkezli d_{ab} yarıçapında bir çember olsun. L_a doğrusunu, C_b'ye q_a'da değen bir eğim ve L_b'yi de C_a'ya q_b'de değen bir eğim doğrusu olarak tanımlayalım, ve L ise q_a ile q_b arasında bir doğru olsun. Figür 2.1'e bakınız. Eğim doğrularının tanımı gereği, $L_a \perp L$ ve $L_b \perp L$'dir. Bu nedenle, L_a ve L_b birbirine paraleldir. Şimdi L_a ve L_b'nin destek doğrusu olduklarını iddia edelim.

p ∈ P ve p ≠ q_a olacak şekilde herhangi bir noktayı seçelim. p noktası, çapın tanımı gereği uzaklık(q_b, p) ≤ d_{ab} olduğundan, C_b'nin ya içindedir ya da üzerindedir. Bu durum, P'nin q_a hariç diğer tüm noktalarında geçerli olduğu için, $L_a \cap P = q_a$'dır ve bu nedenle L_a, P için bir destek doğrusudur. Benzer şekilde, L_b de P için bir destek doğrusudur.

L_a, L_a paralel olduklarından ve her ikisi de destek doğrusu olduğundan, ve

$$\text{çap}(P) = d_{ab} = \text{uzaklık}(L_a, L_b)$$

olduğundan P'nin çapının paralel destek doğruları tarafından belirlendiği sonucuna varırız.



Figür 2.1

Son olarak, uzaklık(L_1, L_2) > d_{ab} olacak şekilde, birbirine paralel olan L_1 ve L_2'nin destek doğruları olduğunu varsayalım. L_1 ve L_2, P ile q_1 ve q_2'de kesişsin. Bu durumda uzaklık(q_1, q_2) ≥ uzaklık(L_1, L_2) > d_{ab} olur, ki bu da bizim d_{ab} = çap(P) varsayımımızla çatışır. Bu nedenle, uzaklık(L_a, L_b) P'nin paralel destek doğruları arasındaki maksimum mesafedir, ve P'nin çapının birbirine paralel destek doğruları arasındaki en büyük uzaklık olduğunu elde ederiz.

Corollary 2.1.2 P konveks çokgeninin çapı, P'nin antipodal nokta çifleri arasındaki uzaklıklardan en büyüğüdür.

> **Kanıt:** Herhangi bir çap çiftinin (diameter pair) antipodal çift olduğu gösterilmişti. Hiçbir antipodal çift, köşeler arasında, çaptan daha büyük bir mesafeye sahip olamayacağından çap, antipodal çiftler arasındaki uzaklıkların en büyüğüdür.

Yukarıda, Hormoz Pirzadeh'in tezinden alınan tanımlar, teoremler ve kanıtlar gösteriyor ki, bir çokgenin tüm antipodal çiftlerini bulan bir algoritma tasarlanırsa, basit bir karşılaştırmayla bu çiftler arasından birbirine en büyük mesafeye sahip olan nokta çifti, çap olarak bulunabilir.

Aşağıdaki algoritma tüm antipodal nokta çiftlerini tespit eder:

// Başlangıç (Initialization) adımı

1) En büyük y değerine sahip köşe noktasını bul.

2) En küçük y değerine sahip köşe noktasını bul.

3) Bunlardan geçecek ve x eksenine paralel olacak şekilde iki kumpası yerleştir.

// Diğer antipodal çiftleri bulma süreci

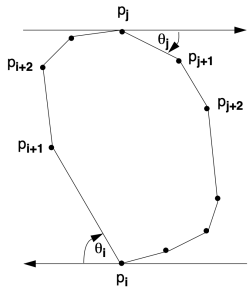
4) İlk kumpas Pi ikincisi ise Pj noktasında çokgenle keşistiğini varsayalım. İlk kumpasın Pi-Pi+1 kenarı ile arasındaki açı ve ikinci kumpasın Pj-Pj+1 kenarı ile arasındaki açı karşılaştırılır (Pj+1 Pj'nin, Pi+1 de Pi'nin saat yönündeki komşusudur. Figür 3.1'e bakınız.) Ve küçük açı belirlenir.

5) Her iki kumpasta saat yönünde bu 'küçük açı' değeri kadar döndürülür. Küçük açılı kumpas tarafından temas edilen yeni köşe noktası ile büyük açılı kumpasın değişmeyen köşe noktası yeni bir antipodal çift olarak eklenir (kumpaslar figür 1.2'deki gibi bir pozisyonda olurlar).

6) Eğer açılar birbirine eşit ise 3 antipodal çift eklenir (kumpaslar figür 1.3'deki pozisyonda olurlar).

7) Döndürüldüğünde yeni köşe noktasına değen kumpasların kesişim noktaları güncellenir (örneğin Pi iken Pi+1 veya Pj iken Pj+1 yapılır.)

8) İlk kumpas Pj'ye ikincisi de Pi'ye ulaşıncaya kadar 4-5-6-7 adımları tekrarlanır.



Figür 3.1

Başlangıç adımı tamamlanmış bir "Dönen Kumpaslar" metodu görselleştirmesi. Bundan sonra kumpaslar Qi ve Qj açılarının değerlerine göre döndürülecekler.

Algoritmanın Tüm Antipodal Çiftleri Elde Ettiğinin İspatı ve Karmaşıklığı

Öncelikle başlangıç adımının doğruluğunu kanıtlayalım. Köşe noktaları verilen P konveks çokgeninde en büyük ve en küçük y değerine sahip noktalar tespit edilip bu noktalardan x eksenine paralel olacak şekilde doğrular geçirildiğinde bunların destek doğrusu olması gerekir, yani çokgenin iç kısmından geçmemeleri gerekir (Tanım 2.1.1). En büyük y değerine sahip noktanın bu koşulu sağlayıp sağlamadığını kontrol edelim. Eğer y_{max} noktasının hem sağındaki hem de solundaki nokta y_{max}'ın değerinden küçük bir y değerine sahipse y_{max}'tan çizilen doğru çokgenle sadece tek bir noktada kesişir ve destek doğrusu olur. Eğer y_{max}'ın sağında veya solunda onunla aynı y değerine sahip bir nokta varsa, çizilen doğru çokgenin bir kenarıyla çakışır ve yine destek doğrusudur. Y_{max}'ın sağında veya solunda ondan daha büyük y değerine sahip bir nokta olamaz, eğer olsaydı y_{max} o olması gerekirdi. Bu durumda ortaya çıkar ki, en büyük nokta seçildiğinde ve x eksenine paralel bir doğru çizildiğinde bu doğru bir destek doğrusudur. Aynı durum en küçük nokta için de geçerlidir. Başlangıç için birbirine ve x eksenine paralel iki doğru bu şekilde edilebilir. En büyük ve en küçük değerlerin bulunması O(n) karmaşıklıkla—her bir karşılaştırma O(1) zaman alır ve n karşılaştırma gerçekleştirilir—gerçekleştirilebilir.

Başlangıç durumunda, birbirine paralel pozisyonda bulunan iki kumpas bulunur. Figür 3.1'e bakınız. Bu durumda biri p_j'den diğeri de p_i'den geçen iki destek doğrusu (kumpas) vardır. Daha sonra, üstteki kumpasın p_j-p_{j+1} kenarı ile arasındaki açı olan Qj ile alttaki kumpasın p_i-p_{i+1} kenarı ile arasındaki açı olan Qi karşılaştırılır. Olası durumlar şunlardır: Qi > Qj , Qi = Qj ve Qi < Qj. Qi > Qj için, her iki kumpası da Qi kadar döndürürsek bu durumda üstteki kumpas çokgenin iç kısmından geçer, destek doğrusu olmayı yitirir. Aynı durum, Qi<Qj için Qj kadar döndürme gerçekleştirildiğinde de geçerlidir (bu durumda alttaki kumpas çokgenin iç kısmından geçer). Genelleştirilecek olursa, bu iki açıdan küçük olanından daha fazla derecede gerçekleştirilecek her döndürme sonucunda çokgenin iç kısmından geçen bir kumpas olur ve destek doğrusu olma özelliğini yitirir. Qi > Qj için Qj, Qi < Qj için de Qi kadar döndürme gerçekleştirildiğinde, genelleştirilecek olursa, kumpaslar, iki açıdan küçük olanı kadar döndürüldüklerinde küçük açıya sahip olan kumpas yeni bir köşe noktasına temas eder (çokgenin bir kenarıyla çakışık olur), büyük açılı olan ise açısı küçülmekle beraber aynı noktada kalmayı sürdürür. Qi = Qj olduğu durumlarda da kumpaslar bu açı değerleri kadar döndürülür ancak 3 yeni antipodal nokta eklenir. Döndürme tamamlandıktan sonra yeni bir köşe noktasına değen kumpasların konumu güncellenir. Başlangıçta kumpaslar paralel konumdaydılar ve süreç içerisinde de aynı derece de döndürüldüler, bu nedenle son durumda da iki kumpas birbirine paraleldir. Birbirine paralel destek doğruları olduklarından yeni bir antipodal nokta çifti elde edilir. Sürecin sonunda, başlangıç durumdaki şartlar devam ettiğinden (paralel destek doğrusu olma), kumpasların yeni konumlarında da bu süreç tekrar edilebilir. Tüm antipodal çifleri elde etmek için, hem üstteki kumpas alttakinin başlangıç noktasına, hem de alttaki kumpas üsttekinin başlangıç noktasına ulaşana kadar tekrar edilir. Her bir döndürmeden sonra en az bir kumpasın çokgenin bir kenarıyla çakışık olacağı kesindir, bu nedenle tam tur tamamlanana kadar tüm köşelere temas edilmiş olur, ve tüm antipodal nokta çiftleri elde edilir. Her bir döndürme, basit bir açı hesabı için işlemler ve bir küçük olanın tespiti için yapılacak bir karşılaştırma ile O(1) zamanda gerçekleştirilebilir. Her iki kumpas da O(n) kez döndürülebilir. Bu durumda, bu kısmın karmaşıklığı da O(n)'dir.

Başlangıç durumunu elde etmenin karmaşıklığı O(n) ve daha sonra tüm antipodal çiftleri elde etmenin karmaşıklığı da O(n) olduğundan algoritmanın toplam karmaşıklığı da O(n) olur, ve bu problem için optimal (en uygun) çalışma süresini verir.

Algoritmanın Rakipleri ve Kısıtları

Algoritmanın rakipleri olarak şunlar sıralanabilir:

1) Basit yaklaşım
2) Snyder and Tang Algoritması
3) Dobkin and Snyder Algoritması

Synder and Tang algoritması, W.E. Snyder ve D. A. Tang tarafından "Finding the extrema of a region" adlı çalışmada yayınlanmıştır. Bu çalışmada, algoritmanın O(n) çalışma zamanına sahip olduğu belirtilir. Ancak daha sonra Binay K. Bhattacharya ve Godfried T. Toussaint tarafından yapılan çalışmada ise bu algoritmanın karşıt örnekleri yayınlanmıştır. Bu karşıt örnekler—örneğin oval şeklindeki çokgenler—algoritmanın bazı tarzdaki çokgenlerde doğru yanıtı vermeyebileceğini göstermiştir.

Dobkin ve Snyder algoritması da O(n) çalışma zamanına sahiptir. Ancak bu algoritmaya ait karşıt örnekler de yayınlanmıştır. Algoritma, konveks çokgenlerin unimodal olduğu varsayımına dayandığından ve bu varsayım da yanlış olduğundan bu tarz örneklerde doğru cevabı vermez.

Basit yaklaşım ise, her durumda doğru cevabı vermekle beraber $O(n^2)$ karmaşıklığa sahip olduğundan verimsizdir. Ancak eldeki veri, Shamos'un algoritmasının başlangıç varsayımlarını karşılamıyorsa ön işlem olmadan kullanılamaz. Bu durumda sadece basit yaklaşım doğru cevabı verir.

Sonuç olarak, Shamos'un algoritması O(n) karmaşıklığından ve çokgenin köşe koordinatları sıralı şekilde verildiği her örnek için doğru sonucu verdiğinden dolayı rakiplerinden üstündür. Ancak eğer köşe koordinatları sıralı şekilde verilmez ise veya sadece köşe koordinatları değil bir noktalar kümesi verilirse bu algoritma kullanılamaz ve çözüm sadece basit yaklaşım ile elde edilebilir. Böyle durumlarda, algoritmayı kullanabilmek için, bir Convex Hull algoritması kullanılabilir ve bir konveks çokgen oluşturulabilir. Ancak en verimli Convex Hull algoritması O(nlogn) karmaşıklığa sahiptir, bu nedenle O(n) karmaşıklık kaybedilir.

Algoritmanın Kullanım Alanları

M. Shamos'un bu algoritmayı bir konveks çokgenin çapını hesaplamak için kullandığından bahsedilmişti. Ancak daha sonra Toussaint tarafından yapılan çalışmalar gösterdi ki, Shamos'un çalışmasında yayınladığı algoritma sadece çap bulmak için kullanılsa da bir çokgen etrafında dönen doğrular fikri genelleştirilip farklı alanlardaki birçok problemi çözmek için kullanılabilir. Dönen kumpaslar metodunun doğru cevabı verdiği farklı problemler Toussaint'in "Solving Geometric Problems with the Rotating Calipers " adlı çalışmasında gösterilmiştir. Toussaint, Shamos'un bulduğu bu metodun iki şekilde genelleştirilebileceğini söyler: bir çokgen üzerinde birden çok kumpas kullanılabilir veya birden çok çokgen üzerinde bir kumpas kullanılabilir. Ayrıca, bu genelleştirmelerin konveks çokgenler üzerinde tanımlı birçok geometrik problem için basit O(n) çözümler sunduğunu gösterir. Çözüm sunulan problemleri şöyle sıralar: çevreleyen en küçük alanlı dikdörtgeni bulmak, iki konveks çokgenin

vektörel toplamı, konveks çokgenlerin birleştirilmesi, ve lineer olarak ayrılabilen kümelerin kritik destek doğrularını bulunması. Son problem, aslında, görünürlük, çarpışma önleme, range fitting, lineer ayrıştırılabilirlik ve kümeler arasındaki Grenander uzaklığının hesaplanması gibi birçok problemin çözümünde uygulanabilir.

Daha sonra, Toussaint'in doktora öğrencisi Hormoz Pirzadeh tarafından yapılan "Computational Geometry with Rotating Calipers" adlı çalışma ise metodun eksik olan kanıtlarını içerir ve bu metotla çözülebilecek yeni problemler ekler. Pirzadeh tarafından yapılan çalışmaya göre dönen kumpaslar metodu kullanılarak şu problemler çözülebilir:

1. Uzaklık ölçümleri
    (a) Konveks bir çokgenin çapı.
    (b) Konveks bir çokgenin genişliği.
    (c) İki konveks çokgen arasındaki maksimum uzaklık.
    (d) İki konveks çokgen arasındaki minimum uzaklık.

2. Çevreleyen Dikdörtgenler
    (a) En küçük alana sahip çevreleyen dikdörtgen.
    (b) En küçük çevreye sahip çevreleyen dikdörtgen.

3. Triangulation ve Quadrangulation
    (a) Onion triangulation.
    (b) Spiral triangulation.
    (c) Quadrangulation.

4. Konveks çokgenlerin özellikleri
    (a) İki convex hull'un birleştirilmesi.
    (b) Kritik destek doğruları.
    (c) Ortak eğimler.
    (d) Vektör toplamları.

Algoritmanın çözüm olarak kullanılabileceği problemlerin sayısının oldukça fazla olduğu görülebilir. Algoritmanın, "Computational Geometry" alanındaki önemi Pirzadeh'in çalışmasındaki şu kısımdan anlaşılabilir:

"Computational geometry is the study of geometric problems and the algorithms to solve them. Problems range from the straigtforward to the very complex in their concept. However, even the least complicated problems rarely have simple algorithms to solve them efficiently. Usually, the best algorithm to solve a given problem is "tailor-made" to suit that specific problem. It is therefore rare to find a paradigm in computational geometry, i. e., a general method or approach to solve a variety of problems. The Rotating Calipers is such a paradigm. As well as being polyvalent, this method's other advantages are its simplicity in concept and its efficiency."

Kaynaklar

- "Computational Geometry", Michael Shamos (1978)
- "Solving geometric problems with the rotating calipers", Godfried T. Toussaint (1983)
- "Computational geometry with the rotating calipers", Hormoz Pirzadeh (1999)
- http://www.personal.kent.edu/~rmuhamma/Compgeometry/MyCG/Definitions/definition.htm
- http://www-cgrl.cs.mcgill.ca/~godfried/research/calipers.html
- https://www.tvhoang.com/articles/2018/12/rotating-calipers
- https://en.wikipedia.org/wiki/Rotating_calipers

```c
//
//  rotatingCalipers.h
//  Rotating Calipers
//
//  Created by Mert Arıcan on 12.06.2020.
//  Copyright © 2020 Mert Arıcan. All rights reserved.
//

#ifndef rotatingCalipers_h
#define rotatingCalipers_h

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>
#include <time.h>

#define PI 3.14159265
float const D = 180.0 / PI;

// Create 'Point' struct to represent points of polygon
typedef struct point {
    int ID;
    char label;
    float x;
    float y;
} Point;

// Create 'Caliper' struct to represent calipers
typedef struct caliper {
    int initialPointID;
    int currentPointID;
    float degree;
    float diffDegree;
    bool completedPath;
} Caliper;

void createNSidedPolygon(Point *polygon, int n);

void createSpecialCase(struct point *polygon) {
    // This function creates special polygon for describing how the
     algorithm works.
    struct point A;
    A.label = 'A';
    A.x = 5;
    A.y = 7;
    A.ID = 0;
    polygon[0] = A;

    struct point B;
    B.label = 'B';
```

```
    B.x = 3;
    B.y = 6;
    B.ID = 1;
    polygon[1] = B;

    struct point C;
    C.label = 'C';
    C.x = 2;
    C.y = 5;
    C.ID = 2;
    polygon[2] = C;

    struct point D;
    D.label = 'D';
    D.x = 3;
    D.y = 2;
    D.ID = 3;
    polygon[3] = D;

    struct point E;
    E.label = 'E';
    E.x = 7;
    E.y = 1;
    E.ID = 4;
    polygon[4] = E;

    struct point F;
    F.label = 'F';
    F.x = 8;
    F.y = 2;
    F.ID = 5;
    polygon[5] = F;

    struct point G;
    G.label = 'G';
    G.x = 9;
    G.y = 5;
    G.ID = 6;
    polygon[6] = G;
}

#endif /* rotatingCalipers_h */
```

```c
//
//  rotatingCalipers.c
//  Rotating Calipers
//
//  Created by Mert Arıcan on 12.06.2020.
//  Copyright © 2020 Mert Arıcan. All rights reserved.
//

#include "rotatingCalipers.h"

// Stores the number of edges that the polygon has.
int N = 1;

// Stores the points with the highest and lowest y values.
int minID = 0;
int maxID = 0;

// Polygon is an array of 'Point's
// These points should be counter-clockwise order and in standart
 form.
Point *polygon;

// For convenience.
bool toDisplay = true;

// For determining diameter.
double maxDist = 0.0;
char bestPair[5];

// MARK: Calculating the angle of a line given two points

float getArctan(Point p1, Point p2) {
    // Given two points calculate the POSITIVE arctan(y/x)
    // Later we will use this value together with quadrant of the line
    // to calculate exact angle.
    if (p1.x == p2.x || p1.y == p2.y) { return 0.0; }
    float y = fabs(p2.y - p1.y);
    float x = fabs(p2.x - p1.x);
    float tan = y / x;
    float arctan = atan(tan) * D;
    return arctan;
}

float getDegree(Point p1, Point p2) {
    // Calculates and returns the angle of line that passes through
     given points 'p1' and 'p2'.
    // First function will calculate the arctan value, then depending
     on the quadrant of the line
    // it will calculate and return the exact angle.

    // Get arctan value.
```

```
    float alpha = getArctan(p1, p2);

    // Calculate and return exact angle.
    if ((p2.y - p1.y) == 0 && (p2.x - p1.x) > 0) { return 0.0; }
    if ((p2.y - p1.y) > 0 && (p2.x - p1.x) > 0) { return alpha; }
    if ((p2.y - p1.y) > 0 && (p2.x - p1.x) == 0) { return 90.0; }
    if ((p2.y - p1.y) > 0 && (p2.x - p1.x) < 0) { return 180 - alpha;
     }
    if ((p2.y - p1.y) == 0 && (p2.x - p1.x) < 0) { return 180.0; }
    if ((p2.y - p1.y) < 0 && (p2.x - p1.x) < 0) { return 180.0 +
     alpha; }
    if ((p2.y - p1.y) < 0 && (p2.x - p1.x) == 0) { return 270.0; }
    if ((p2.y - p1.y) < 0 && (p2.x - p1.x) > 0) { return 360.0 -
     alpha; }

    return 0.0;
}

float getDegreeOfASegmentWithStartingPoint(Point p) {
    // Calculates and returns the degree of the segment (edge) of the
     polygon that starts with given point 'p'.
    return getDegree(polygon[p.ID], polygon[(p.ID+1)%N]);
}

float calculateDistanceBetween(Point p1, Point p2) {
    // Calculates and returns the Euclidean distance between two
     points.
    return sqrtf(powf((p1.x - p2.x), 2) + powf((p1.y - p2.y), 2));
}

void findInitialAntipodals(Point *polygon) {
    // Finds highest and lowest points on the polygon.
    // Assigns these points to 'minID' and 'maxID' variables.
    float minY = 1000000.0;
    float maxY = -1000000.0;
    for (int i = 0; i < N; i++) {
        if (polygon[i].y < minY) {
            minY = polygon[i].y;
            minID = polygon[i].ID;
        }
        if (polygon[i].y > maxY) {
            maxY = polygon[i].y;
            maxID = polygon[i].ID;
        }
    }
}

void initializeCalipers(Caliper *L, Caliper *U) {
    // Puts a lower caliper on the lowest point of the polygon and
     sets its degree to 0,
```

```
    // and puts a upper caliper on highest point of the polygon and
     sets its degree to 180.
    L->initialPointID = minID;
    L->currentPointID = minID;
    L->degree = 0.0;
    L->diffDegree =
     getDegreeOfASegmentWithStartingPoint(polygon[minID]) - 0.0;
    L->completedPath = false;

    U->initialPointID = maxID;
    U->currentPointID = maxID;
    U->degree = 180.0;
    U->diffDegree =
     getDegreeOfASegmentWithStartingPoint(polygon[maxID]) - 180.0;
    U->completedPath = false;
}

void addAntipodalPairs(int first, int second, int a, int b) {
    // Depending on the 'a' and 'b' values, this function adds 4, 2
     or 1 antipodal pairs.
    // exp: If 'a' is 1 then that means not just add 'first' point
     but add 'first+1'th point also
    // as an antipodal pair. This is same for 'b' and 'second'.
    int temp = b;
    while (a >= 0) {
        while (b >= 0) {
            if (!(a == 1 && b == 1) && toDisplay) {
                struct point p1 = polygon[(first+a) % N];
                struct point p2 = polygon[(second+b) % N];
                if (calculateDistanceBetween(p1, p2) > maxDist) {
                    sprintf(bestPair, "%c-%c", p1.label, p2.label);
                    maxDist = calculateDistanceBetween(p1, p2);
                }
                printf("pair: %c-%c\n", p1.label, p2.label);
            } b--;
        }
        b = temp; a--;
    }
}

void updateCalipersAfterRotation(Caliper *L, Caliper *U) {
    // Makes necessary updates for calipers after rotation.
    L->currentPointID = L->currentPointID % N;
    U->currentPointID = U->currentPointID % N;

    // If any of the degrees is greater than 360.0 then subtract
     360.0 from it.
    // Modulo doesn't work for float values.
    if (L->degree > 360.0 ) { L->degree = L->degree - 360.0; }
    if (U->degree > 360.0 ) { U->degree = U->degree - 360.0; }
```

```
    // When upper caliper reaches the lower calipers initial point or
     vice versa
    // that means caliper completed its path. When both complete all
     procedure stops.
    if (L->currentPointID == U->initialPointID) {
        L->completedPath = true;
    }
    if (U->currentPointID == L->initialPointID) {
        U->completedPath = true;
    }

}

void rotatingCalipers(Caliper *L, Caliper *U) {
    // Main procedure of the process of calculating the diameter of a
     convex polygon.

    while (!L->completedPath || !U->completedPath) {
        L->diffDegree =
         fabs(getDegreeOfASegmentWithStartingPoint(polygon
         [L->currentPointID]) - L->degree);
        U->diffDegree =
         fabs(getDegreeOfASegmentWithStartingPoint(polygon
         [U->currentPointID]) - U->degree);

        // If next segments (edges) are parallel with each other...
        // Also means that both of the calipers overlap with polygons
         edges
        if (L->diffDegree == U->diffDegree) {
            addAntipodalPairs(L->currentPointID, U->currentPointID,
             1, 1);
            L->currentPointID++;
            U->currentPointID++;
            L->degree += L->diffDegree;
            U->degree += U->diffDegree;
        }
        // If one of the calipers is overlaps with one of the edges
         of a polygon...
        else if ( L->diffDegree == 0.0 || U->diffDegree == 0.0 ) {
            Caliper *tmp1 = (L->diffDegree == 0) ? L : U;
            Caliper *tmp2 = (tmp1 == L) ? U : L;
            addAntipodalPairs(tmp1->currentPointID,
             tmp2->currentPointID, 1, 0);
            tmp1->currentPointID++;
        }
        // If "difference with the next segment" is different for two
         calipers...
        else {
            Caliper *far = (L->diffDegree >= U->diffDegree) ? L : U;
            Caliper *close = (L->diffDegree < U->diffDegree) ? L : U;
```

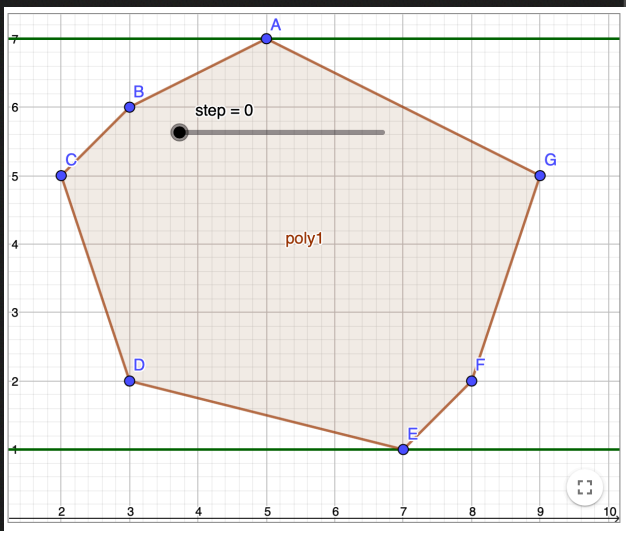
```
            float difference = fabs(far->diffDegree -
             close->diffDegree);
            far->degree += difference;
            close->degree += difference;
            addAntipodalPairs(far->currentPointID,
             close->currentPointID, 0, 0);
            close->currentPointID++;
        }
        updateCalipersAfterRotation(L, U);
    }
}

void complexityAnalysis() {
    // Function for printing the time used by the 'rotatingCalipers'
     procedure with different sized inputs
    // And drawing bar diagram for the results.

    // Initialize variables for storing the starting and ending time
     of the process
    clock_t start, end;

    // Set 'toDisplay' value to 'false' to indicate that we don't
     want results to get printed on screen.
    toDisplay = false;

    // Allocate memory for polygon
    polygon = malloc(sizeof(Point)*2);

    // Initialize variable for storing the time of the process.
    double cpuTimeUsed;

    // Create variable 'x' for determining maximum coefficient (max
     is 'x-1').
    int x = 11;

    // Initialize an array of size 'x' for storing values from index
     1 to 'x-1'.
    // Starts from 1 because we don't need the coefficient value of
     zero in the loop below.
    int results[x];

    double unit = 0.15;

    for (int i = 1; i <= x-1; i++) {
        // for every coefficient from 1 to x-1...

        // Determine the number of sides of the polygon
        N = 1000000 * i;

        // Reallocate memory for polygon with that size.
        Point *temp = realloc(polygon, sizeof(Point)*N);
```

```
    polygon = temp;

    // Create the polygon
    createNSidedPolygon(polygon, N);

    // Allocate memory for calipers
    Caliper *L = malloc(sizeof(Caliper));
    Caliper *U = malloc(sizeof(Caliper));

    // Find initial antipodal pairs for beginning of the process
    findInitialAntipodals(polygon);

    // Set appropriate initial values for calipers
    initializeCalipers(L, U);

    // Get time when procedure starts
    start = clock();

    // Start procedure
    rotatingCalipers(L, U);

    // Get time when procedure ends
    end = clock();

    // Free memory used by calipers
    free(L); free(U);

    // Calculate the time used by process
    cpuTimeUsed = ((double) (end - start)) / CLOCKS_PER_SEC;

    // Determine first 'cpuTimeUsed' as unit time.
    if (i == 1) { unit = cpuTimeUsed; }

    // Store the time in results array
    results[i] = (int) (cpuTimeUsed / unit);

    // Prinf time info to the consol
    printf("For n: %d, time used: %f\n", 1000000 * i,
     cpuTimeUsed);
}
printf("\nUnit time: %f seconds\n\n", unit);

// MARK: Drawing bar diagram
for (int i = 1; i <= x-1; i++) {
    printf("%d ", 1000000 * i);

    // For the symmetry of the bar diagram
    if (i != x-1) { printf(" "); }

    for (int j = 0; j < results[i]; j++) {
        printf("🟧");
```

```
        }
        printf(" %d\n", results[i]);
    }
    free(polygon);
}

void createNSidedPolygon(Point *polygon, int n) {
    // Function for constructing the corners of a regular 'n' sided
     polygon.
    double radius = 100.0;
    char label = 'A';
    for (int i = 0; i < n; i++) {
        struct point p;
        p.ID = n-1-i;
        p.label = label+i;
        p.x = sin((double) i / (double) n * 2.0 * PI) * radius;
        p.y = cos((double) i / (double) n * 2.0 * PI) * radius;
        polygon[n-i-1] = p;
    }
}

int main(void) {

//    MARK: Special cases for testing the validity of results.
    N = 7;
    polygon = malloc(sizeof(Point)*N);
    if (polygon == NULL) { return 1; }

    if (N == 7) { createSpecialCase(polygon); }
    else { printf("N must be set to 7 before calling
     'createSpecialCase' function.\n"); return 1;}
//    createNSidedPolygon(polygon, N);

    Caliper *L = malloc(sizeof(Caliper));
    Caliper *U = malloc(sizeof(Caliper));
    if (L == NULL || U == NULL) { return 1; }

    findInitialAntipodals(polygon);
    initializeCalipers(L, U);

    rotatingCalipers(L, U);

    free(polygon);
    free(L);
    free(U);
    printf("Diameter: %f, pairs: %s\n", maxDist, bestPair);

//    MARK: Complexity analysis
//    complexityAnalysis();
    return 0;
}
```

Ekran Görüntüleri

```
Rotating Calipers — -zsh — 80×24
Last login: Thu Jun 18 10:12:03 on ttys001
mertarican@mert-mbp Rotating Calipers % gcc rotatingCalipers.c
mertarican@mert-mbp Rotating Calipers % ./a.out
pair: E-A
pair: F-B
pair: E-C
pair: E-B
pair: C-F
pair: G-C
pair: D-G
pair: A-D
Diameter: 7.000000, pairs: G-C

mertarican@mert-mbp Rotating Calipers %
```



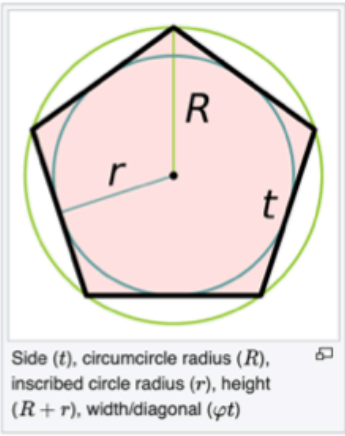
Kod, yandaki çokgen için doğru cevabı yazdırıyor.

```
Rotating Calipers — -zsh — 80×24
Last login: Thu Jun 18 10:15:37 on ttys001
mertarican@mert-mbp Rotating Calipers % gcc rotatingCalipers.c
mertarican@mert-mbp Rotating Calipers % ./a.out
Diameter of a regular pentagon on circle of radius 100.0
Diameter: 190.211319, pairs: E-C

mertarican@mert-mbp Rotating Calipers %
```

$$\text{Diagonal} = R\sqrt{\frac{5+\sqrt{5}}{2}} = 2R\cos 18^\circ = 2R\cos\frac{\pi}{10} \approx 1.902R,$$

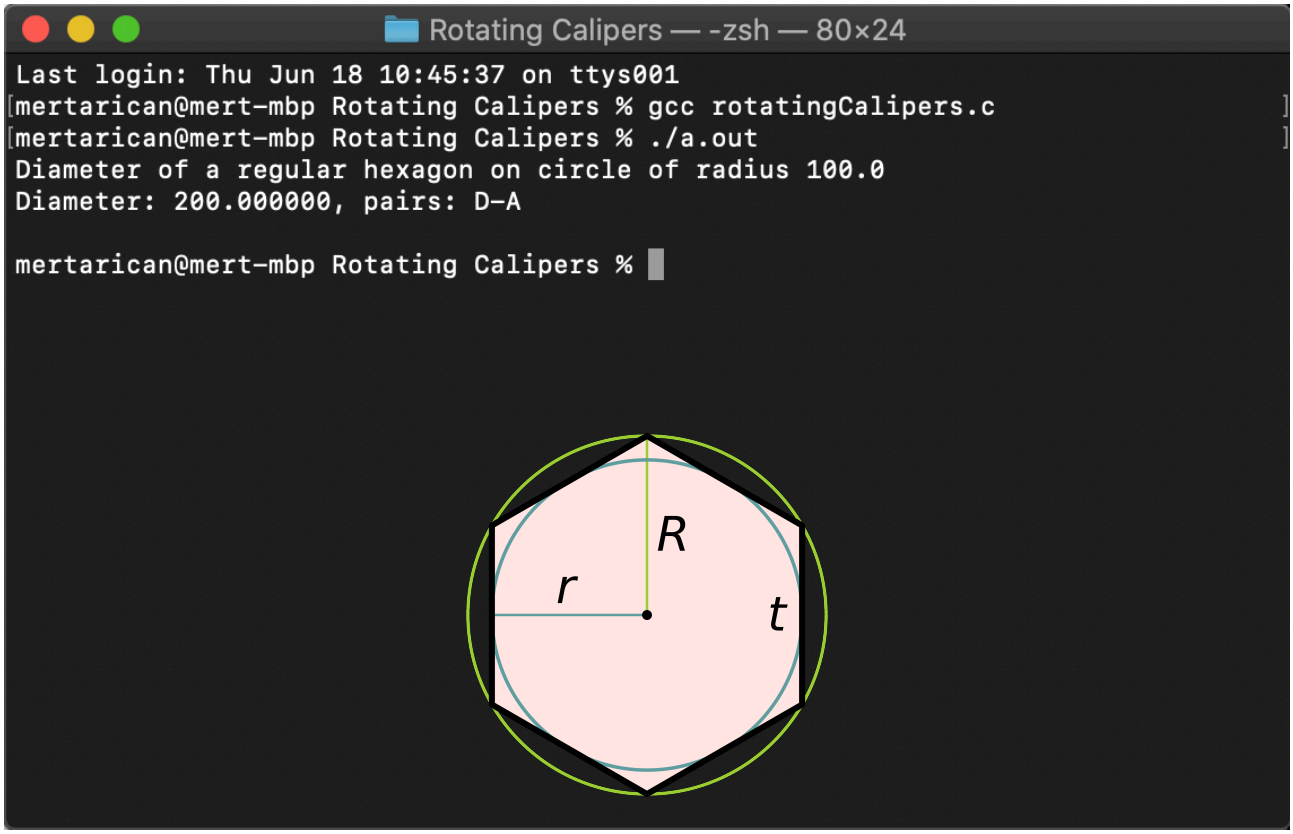
where R is the radius of the circumcircle.

The area of a convex regular pentagon with side length t is given by

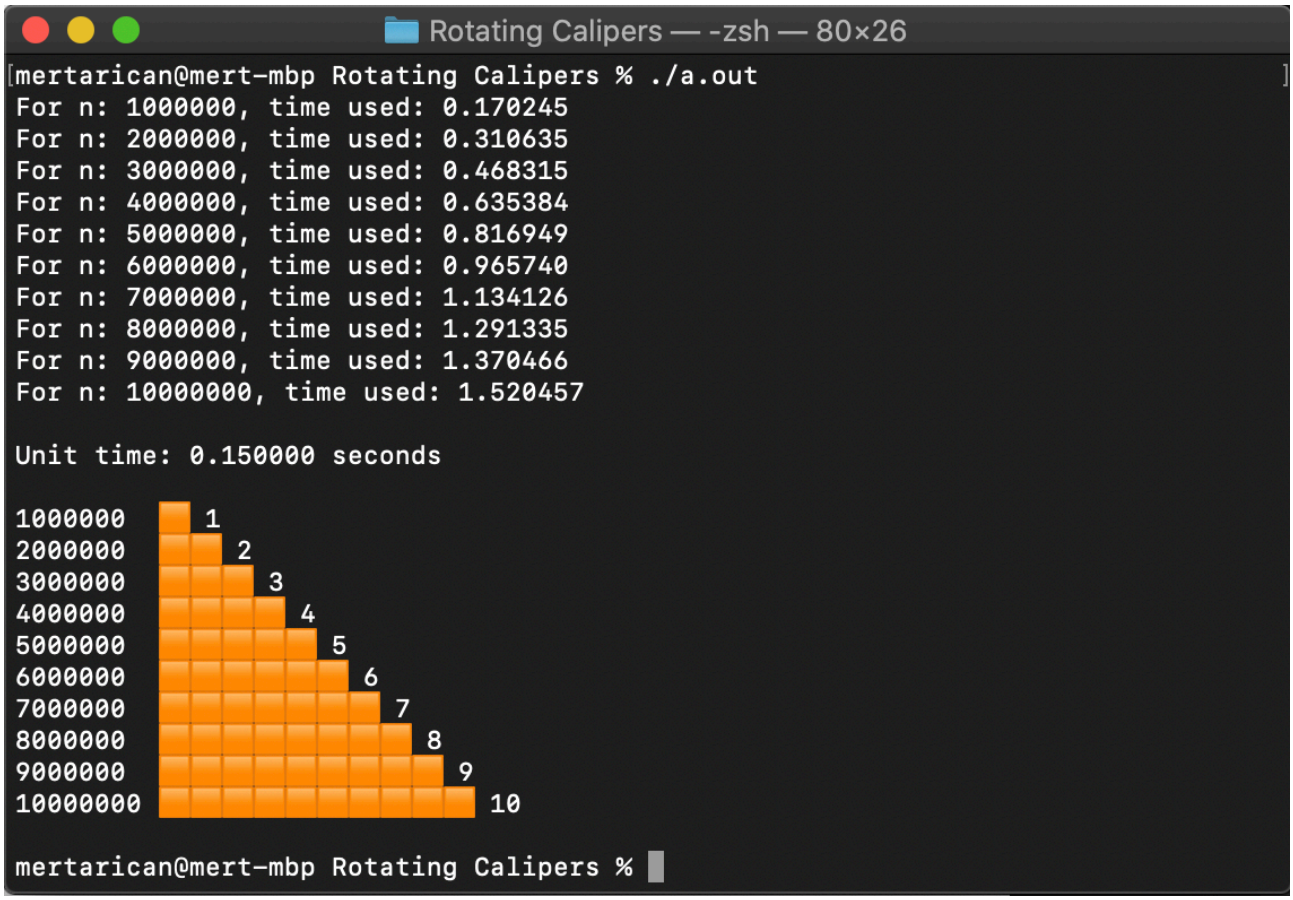


Side (t), circumcircle radius (R), inscribed circle radius (r), height ($R + r$), width/diagonal (φt)

Wikipedia'da pentagon başlığındaki denklem sonucu doğruluyor. Köşeleri 100.0 birim yarıçaplı çemberin üzerinde bulunan beşgenin çapı ≈ 190.2.



Köşeleri 100 birim yarıçaplı çember üzerinde bulunan düzgün altıgenin çapı için verilen sonuç. Sonuç, Wikipedia'da "Hexagon" başlığındaki görsel tarafından doğrulanıyor.



Sırasıyla bir milyon, iki milyon, …, on milyon kenarlı çokgen için kod çalıştırıldığında her birinin çalışma süresi yukarıdaki gibidir. İlk çalışma süresine yakın bir değer birim süre olarak seçilirse yukarıdaki grafik elde edilebilir. Grafik, O(n) karmaşıklığı doğrular. Ancak çalışma süreleri farklı donanımlarda farklı sonuçlar verebileceğinden "Unit time" grafiğin şeklini belirleyen ana unsurlardan biridir, farklı donanımlarda farklı "Unit time" seçilmesi gerekebilir. Regresyon ile daha güvenilir sonuçlar elde edilebilir.