Büyüyen Kısmi Yol Kısıtlarıyla Konkolik Test

Yavuz Köroğlu, Alper Şen

Boğaziçi Üniversitesi, Bilgisayar Mühendisliği Bölümü
{yavuz.koroglu,alper.sen}@boun.edu.tr

Özet. Otomatik birim testler sayesinde programların kapsama oranını arttırmak mümkündür. Konkolik test metodu (concolic testing) otomatik birim test yaratımı için kullanılan bir yazılım testi tekniğidir. Ancak, yol patlaması (path explosion) ve kısıt çözümlerinin (constraint solving) yarattığı darboğaz nedeniyle, ölçeklenebilir bir otomatik konkolik testçi geliştirmek zor olmuştur. Bu tekniği ölçeklenebilir hale getirebilmek için bu makalede kısıt çözücüye daha çok ama daha küçük sorgular gönderilerek çözme üzerindeki yükü hafifletecek bir yol önerilmiştir. Yaptığımız deneyler altta yatan kısıt çözücüye yapılan sorgu uzunluklarını düşürerek bu hedefe yaklaşıldığını gösteriyor.

Anahtar Kelimeler: Konkolik Test (Concolic Testing), Hedef Yönelimli Test (Goal Directed Testing), Dallanma Zorlaması (Branch Enforcement), Kısıt Çözümü (Constraint Solving)

1 Giriş

Konkolik testte amaç test edilen programın yollarını dolaşacak girdileri otomatik olarak oluşturmaktır. Buna örnek olarak bir üçgen sınıflandırma programını düşünelim. Bu program test edilirken bütün olası sonuçların (çeşitkenar, ikizkenar, eşkenar, üçgen değil) bir şekilde ortaya çıkarılmasını sağlayacak test girdileri üretilmesini bekleriz. Tüm yolları kapsayacak şekilde bir test üretmek zor ve uzun vakit alan bir iştir. Bu işin hızlanabilmesi, daha büyük programlarda da otomatik test üretimini mümkün kılacaktır. Konkolik testin daha hızlı çalışması için önerdiğimiz yöntem kısıt çözücüye yapılan sorgu uzunluklarını düşürerek bu amaca hizmet etmeyi öngören bir ön çalışmadır. Konkolik test metodu program çalışmasını (program execution) istenilen yollara yönlendiren girdileri yaratmaya çalışır. Bu girdileri yaratabilmek için, istenilen yolların yol kısıtları (path constraint) bulunur. Böyle kısıtları sağlayan girdileri bulmak kısıt sağlama problemi (constraint satisfaction problem) olarak bilinen çok zor bir problemdir. Bu problemleri çözmek genelde girdinin büyüyüşüyle üssel oranda vakit alır. Bu yüzden kısıt sağlama için yapmayı planladığımız şey mümkünse kısıt çözücüye yapılan sorgu sayısını artırmak pahasına sorguları kısaltmaktır. Bir büyük kısıtın çözümü için daha küçük kısıtların çözümünü kullanmak üzerinde çalışılmış bir fikirdir [1]. Kısıt çözücüleri bu şekilde kullanmak yazılım doğrulaması (verification) alanında önerilmiştir [2]. Yöntemimizi CREST adlı konkolik test programında gerçekleştirdik. Yaptığımız ilk deneysel çalışmalarda daha küçük denemeler ile aynı kapsama oranına ulaşılabildiğini gördük. Makalenin geri

kalanı şu şekilde düzenlenmiştir. Bölüm 2 konkolik test üzerine yapılmış diğer çalışmaları açıklamaktadır. Bölüm 3 konkolik test ve geliştirilmiş yeni yöntem hakkında detaylı bilgi vermektedir. Bölüm 4 ise önerilen metodun yararlı olduğu bir örneği içermektedir. Bölüm 5 ise küçük programlar üzerindeki deneysel sonuçları gösterirken bu sonuçları tartışmaktadır. Bölüm 6 ise şimdiye kadar gelinmiş olan noktayı belirtirken Bölüm 7 ise daha ileride yapılabilecekleri anlatmaktadır.

2 İlgili Çalışmalar

Konkolik test verilen bir programın istenilen kısımlarını çalıştırmak için gerekli girdileri üreten ve programları somut olarak bu girdilerle çalıştıran yazılım testi temelli bir yaklaşımdır. Konkolik test programı istenilen tek bir hedefe yönlendirilebilir [3] veya olası bütün hedeflere varmaya çalışabilir [4] [5] [6]. Java [7] [8], C [5] ve C# [6] gibi bilinen dillerin çoğunda kullanılmıştır. Konkolik test büyük programlarda çalıştırılamadığından bu yöntemin farklı bağlamlarda geliştirilmesi için çalışmalar yapılmıştır [9]. Konkolik testte kullanılan yaklaşımlar üzerine detaylı bir araştırma bulunmaktadır [5]. Geleneksel konkolik testin özünde tüm program yolları üzerinde derinlik öncelikli arama (depth first search) vardır [4]. Bu aramalarda uygulamanın somut çalışımı sonucu hesaplanmış sembolik yollar kullanılır. Bu yaklaşımda yol üzerinde denenmemiş en son dallanma tersine çevrilmeye çalışılır. En çok uygulanan alternatif yaklaşımlardan biri ise genişlik öncelikli aramadır (breadth first search) [1] [10]. Bu yaklaşımda algoritma rastgele bir girdiden başlar ve yol üzerinde denenmemiş ilk dallanmada sapmaya neden olacak girdileri bulmaya çalışır. İlk seviyedeki dallanmalar bittiğinde bir sonraki dallanmalara geçilir ve yeni girdiler oluşturulurken önceki yaratılmış girdilerden faydalanılır. Bu yaklaşım başta çok küçük yol kısıtları yaratır ve kısıtlı bir test yapma yaklaşımına olanak sağlar. Başka bir arama alternatifi de kontrol-akış yönlendirmeli (control-flow directed) testtir [5]. Algoritma programı istatistiki olarak kontrol-akış çizelgesindeki en yakın ifadelere yönlendirmeye çalışır. Program çalışma yolları üzerinde arama yaptıkça algoritmanın yavaşladığı görülmüş, bu yüzden rastgele yeniden başlamalar kullanılmıştır. Rastgele-dallanma test yaklaşımı uygulamanın somut çalışımı sonucu hesaplanmış sembolik yol üzerindeki herhangi bir dallanmayı rastgele olarak seçer [5]. Rastgele-dallanma aramasının yanı sıra, rastgele-yol araması da denenmiştir. Bu yaklaşım rastgele yollar seçmeyi ve çalışmayı bu yollara yönlendirmeyi dener. Bu yaklaşımın yazılım testinde en sık kullanılan rastgele-girdi aramasından daha iyi çalıştığı iddia edilmiştir. Bu yaklaşım somut bir çalışma seçer ve bu çalışma yolu üzerindeki her dallanmayı $\frac{1}{2}$ olasılıkla tersine çevirir. Sonra da bu yeni yol kısıtı için girdi yaratmaya çalışır. Rastgele-dallanma yaklaşımının kapsama bakımından daha iyi çalıştığı gösterilmiştir [5]. Bir diğer ilginç yaklaşım ise hedef-yönelimli dallanma zorlamasıdır (goal-directed branch enforcement). Bu yaklaşımda belirli bir hedef için sembolik yol kısıtları toplanır ve bütün kısıt giderek artan şekilde kritik dallanma koşullarını çözerek sağlanır [11]. Bizim çalışmamızda ise, söz konusu yaklaşımın tüm yolları deneyen metodlara genellenmesi sunulmaktadır. Konkolik testte kullanılan kısıt sağlama problemi verilen kısıtlar içerisinde kalan bir

örnek bulmak olarak tanımlanabilir. Konu hakkında daha fazla bilgi [12] kaynağından elde edilebilir. Yices, Z3 gibi standart kısıt çözücülerin yanı sıra nonlineer kısıtları çözmek [13] ve gerçel sayılarda yüksek kesinlik [14] için ayarlanmış kısıt çözücüler vardır. Kısıt çözümü 3-sağlanabilirlik (3-satisfiability) probleminin bir genellemesi olup problemin büyük örneklerini çözen bir algoritma bulunması beklenmemelidir. Bu makalede kullandığımız yöntem derinlik öncelikli arama ile hedef-yönelimli konkolik test algoritmalarının birleştirilmesiyle geliştirilmiştir.

3 Yöntem

3.1 Konkolik Test

Konkolik (Concolic) test ingilizce **conc**rete (somut) ve symb**olic** (sembolik) testlerin birleşimi için bir kısaltmadır. Bu yaklaşımda, test altındaki program sağlanması gereken dallanma koşullarının (branch condition) sembolik olarak toplanılmasını sağlayacak şekle getirilir. Sonra toplanmış koşullar kullanılarak yeni bir koşul kümesi elde edilir. Bu yeni koşullar incelenilerek yeni girdiler yaratılır ve programa girdi olarak sunulur.

Tanım 1 *DALLANMA KOŞULU (Branch Condition, c) En az bir program parametresine (girdi) bağlı sadece **doğru** veya **yanlış** olabilen ve her farklı sonucun programı farklı bir çalışma yoluna yönlendirdiği koşullardır.*

Dallanma koşulları aşağıdakilerden herhangi biri olabilir:

- Boole cebri (boolean algebra) kullanılarak yazılmış bir ifade,
- Bir karşılaştırma veya
- Sadece **doğru** veya **yanlış** döndüren bir fonksiyon.

Genelde bir programdaki dallanma koşulları birbirine bağlıdır, o yüzden bir koşulun sonucunu sabit tutmak diğerlerini etkiler. Rastgele bir çalışma yolu üzerindeki rastgele bir koşulun, bu yol üzerindeki diğer dallanmaların ortalama $\frac{N}{8}$ i ile bağlı olduğu gösterilmiştir (N bu yol üzerindeki toplam dallanma koşulu sayısı olarak alınmıştır) [4].

Tanım 2 *DALLANMA KOŞULU BAĞLILIĞI (Branch Condition Dependency) İki dallanma koşulu sadece ve sadece aşağıdaki durumlarda **birbirine bağlı** kabul edilir:*

- *İki dallanma koşulunun d gibi en az bir ortak değişkeni varsa veya*
- *İki koşul da üçüncü bir koşulla bağlıysa.*

Olası bir çalışma yolunun gerçeklenebilmesi ancak o yol üzerindeki bütün dallanma koşullarının sağlanmasıyla olabilir.

Tanım 3 *TAM YOL KISITI (Full Path Constraint, π) Tam yol kısıtı, bir çalışma yolu üzerindeki bütün dallanma koşullarının* ***mantıksal Ve*** *operatörüyle birleştirilmesi olarak tanımlanır:*

$$\pi = \bigwedge_{i=1}^{N} c_i$$

Şu andan itibaren **kısıt çözücü** adı verilecek olan ve varsa tam yol kısıtını sağlayan değerler yaratabilen bir çözücünün olduğu varsayılacaktır. Böylece, gerçeklenebilir (feasible) bütün çalışma yollarına götürecek olan gerekli girdiler yaratılıp program bu girdilerle çalıştırılabilir. Örneğin, programı bir **hata** ifadesine yönlendiren bir yol gerçeklenmeye çalışılıyor ve buna karşılık gelen tam yol kısıtı biliniyorsa ya bu ifadeye varacak girdiler yaratılabilir ya da bu çalışma yolunun gerçeklenemez (olanaksız) olduğu sonucuna varılır.

Konkolik testte, program önce **rastgele** girdilerle çalıştırılır. Sonra, çalışma sırasında sembolik çalışma yolunun tam yol kısıtı toplanır. Asıl kısıt üzerinden yeni bir tam yol kısıtı elde etmek için bir **strateji** izlenir. Genel bir strateji bu tam yol kısıtı üzerindeki son dallanma koşulunu tersine çevirmektir [4]. Sonra yeni yol kısıtı **kısıt çözücü**ye verilir ve program için yeni test girdileri elde edilir.

$$\pi' = \neg c_N \wedge \bigwedge_{i=1}^{N-1} c_i \tag{1}$$

Bu stratejiye *derinlik-öncelikli arama* adı verilir. Uygulamalar genel olarak kısıt sayısıyla veya azami iterasyon sayısıyla [5] sınırlandırılmıştır. Böyle bir algoritma Algoritma 1 üzerinde açıklanmıştır.

3.2 Kısmi Yol Kısıtları

Teoride, bir kısıt çözücünün çalışma süresi çözücüye verilen ifade (koşul) sayısıyla üssel olarak büyür [15]. Bu yüzden, kısıt çözücüye büyük kısıtlar vermek kötü bir fikirdir. Konkolik testte, ölçeklendirilebilirliğin önündeki üç darboğaz yol patlaması, yüksek dallanma sayısı ve kısıt çözücüye verilen bu büyük sorgulardır [16]. Bizim yaklaşımımızda, sorgu uzunlukları sorgu sayısını artırmak pahasına düşürülmeye çalışılmıştır.

Tanım 4 *KISMİ YOL KISITI (Partial Path Constraint, ϕ) Tam yol kısıtının her yukarı yaklaşımı (over-approximation) bir* ***kısmi yol kısıtı*** *olarak adlandırılır. Tam yol kısıtını yol üzerindeki dallanma koşullarının bir kümesi olarak düşünürsek, kısmi yol kısıtını da tam yol kısıtının herhangi bir alt kümesi olarak görebiliriz.*

$$\pi \rightarrow \phi$$

Tanıma göre, mutlak doğruluk (true) bütün tam yol kısıtlarının bir yukarı yaklaşımıdır. Ayrıca, çalışma yolu üzerindeki her dallanma koşulunun da bir kısmi yol kısıtı olduğu görülebilir.

Algoritma 1 Derinlik Öncelikli Aramalı ve Azami İterasyonlu Konkolik Test.

Require: P : program under test
Ensure: *inputs* : Test Suite
1: $inputs \leftarrow \emptyset$
2: $input \leftarrow$ generateRandomInputs()
3: **for** $j = 1$ **to** max_iterations **do**
4: Execute P with *input*
5: Add *input* to *inputs*
6: $\pi \leftarrow$ collectedPathConstraint
7: **for** $i =$ length(π) **to** 0 **do**
8: **if** !isVisited(branchOf($\pi[i]$) **then**
9: $c \leftarrow \pi[i]$
10: **break**
11: **end if**
12: **end for**
13: **if** isDefined(c) **then**
14: $\pi[i] \leftarrow \neg c$
15: $input \leftarrow$ generateInput(π)
16: **else**
17: **break**
18: **end if**
19: **end for**

Teorem 1 *YOL KISITLARININ YUKARI YAKLAŞIMI (Over-approximation of Path Constraints)*
Bir tam yol kısıtını sağlayan bütün örnekler ayrıca bu yolun bütün kısmi yol kısıtlarını da sağlar.

Eğer sadece kısmi yol kısıtını çözersek, bu çözümün her zaman tam yol kısıtını sağlama şansı vardır. Bu çalışmada, tam yol kısıtlarının kullanımından kaçınabilmek amacıyla kısmi yol kısıtlarının yaratılması üzerine çalışılmıştır. Burada asıl hedef bütünlük (completeness) ve geçerlilikten (soundness) ödün vermeden büyük kısıtların yarattığı yükü azaltmaktır.

3.3 Kısmi Yol Kısıtı Yaratma Stratejileri

Algoritma 2'da program üzerinde tek bir çalışma yolunu gerçekleyecek girdi üretilmektedir. Bunun için sembolik olarak bu hedefin tam yol kısıtı bilinmelidir. Eğer 3. satırda kısmi yol kısıtını sağlayan girdiler bulunamıyorsa, tam yol kısıtı da sağlanamıyor demektir. Eğer 8. satırda öğrenilen tam yol kısıtı hedef ile aynıysa istenilen girdi elde edilmiş demektir. Eğer hiçbiri değilse, o zaman yolun sapmasına neden olan bir koşul vardır. Bu sapma 15. satırda öğrenilmektedir.

Algoritma 3'te ise Algoritma 2 kullanılarak tüm yollar gerçekleyecek olan bir girdi listesi üretilmektedir. Derinlik öncelikli arama kullanılmıştır. 15. satırda görüldüğü gibi daha önce gerçeklenmiş çalışma yollarından yeni tam yol kısıtları elde edilerek Algoritma 2'ye verilmektedir.

Algoritma 2 Büyüyen Kısmi Yol Kısıtları Kullanan Hedef-Yönelimli Konkolik Test

Require: P : program under test, π : full path constraint of goal
Ensure: $input \rightarrow \pi$

$\phi \leftarrow \pi[\text{length}(\pi) - 1]$
loop
  $input \leftarrow \text{generateInput}(\phi)$
  if $input = infeasible$ **then**
    return *fail*
  end if
  Execute P with $input$
  $\pi' \leftarrow$ collectedPathConstraint
  if $\pi = \pi'$ **then**
    return $input$
  end if
  repeat
    $c \leftarrow \text{nextConditionOf}(\pi)$
  until $!\text{contains}(\pi', c)$
  $\phi \leftarrow \phi \wedge c$
end loop

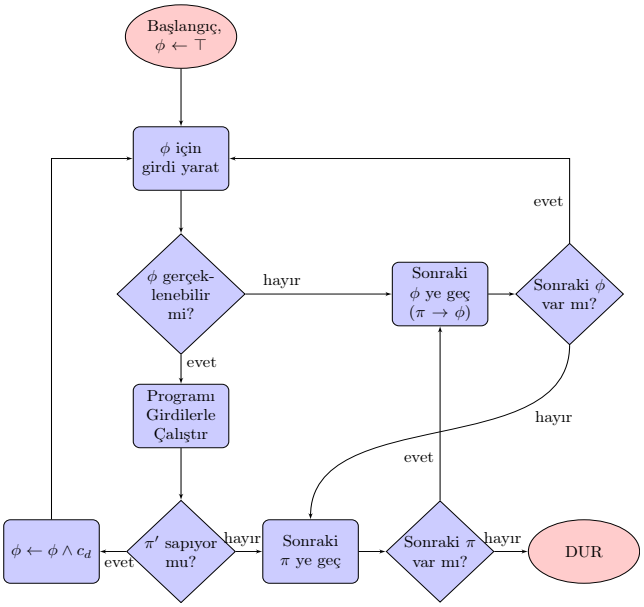
Algoritma 3 Büyüyen Kısmi Yol Kısıtlarını Kullanan Tam Konkolik Test

Require: P : program under test
Ensure: $inputs$: Test Suite

$inputs \leftarrow \emptyset$
$\phi = true.$
$girdi \leftarrow \text{generateInput}(\phi)$
for $j = 1$ **to** max_iterations **do**
  Execute P with $input$
  Add $input$ to $inputs$
  $\pi \leftarrow$ collectedPathConstraint
  for $i = \text{length}(\pi)$ **to** 0 **do**
    if $!\text{isVisited}(\text{branchOf}(\pi[i])$ **then**
      $c \leftarrow \pi[i]$
      break
    end if
  end for
  if $\text{isDefined}(c)$ **then**
    $\pi[i] \leftarrow \neg c$
    $input \leftarrow$ Algortihm 2(P, π)
    Add $input$ to $inputs$
  else
    break
  end if
end for

Teorem 2 *YOL SAPMASI (Path Divergence) Kısmi kısıt ϕ yi sağlayan ama tam yol kısıtı π'yi sağlamayan girdiler var ise bu girdiler için $(\pi \rightarrow c) \wedge (\pi' \rightarrow \neg c)$ ifadesini sağlayan ve yol sapmasına neden olan c vardır. Bu ifadede π', ϕ yi sağlayan girdilerin karşılık geldiği çalışma yolunun tam yol kısıtını belirtmektedir.*

Algoritma 3'ün perfromansını arttırmak için 15. satırda daha uzun ama kısıt çözücüye yüklenmeyen bir ϕ ile başlanabilir. Bu sefer ϕ yine son dallanma koşulunun tersi $\neg c_N$ i içerir, ama bundan başka birbirinden bağımsız (Tanım 2) diğer bütün dallanma koşullarını da içerir. Böyle bir ϕ yi çözmek, kısıt çözücü için $\phi = c_N$ yi çözmek kadar kolay olacaktır, çünkü kısıt çözücü böyle bir durumda her dallanma koşulunu ayrı ayrı çözer [4] [17]. Bu durumda kısmi yol kısıtları daha çok değişken içerdiği için çalışma yolunu sağlama şansı daha yüksek olacaktır.

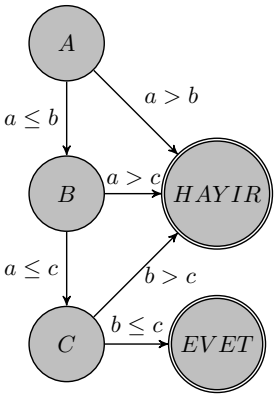


Şekil 1. Büyüyen Kısmi Yol Kısıtlı Konkolik Test Akış Çizelgesi

Görsel kolaylık vermek için Şekil 1 te geliştiridiğimiz algoritmanın akış çizelgesi vardır. Bir π listesi olduğunu ve de her π için bütün yolları deneyen bir ϕ listesi olduğunu varsaymaktadır. Genel olarak sonsuz yol olabilir ve azami iterasyon sınırı durma koşulu olarak kullanılır. Akış çizelgesi ayrıca c_d değişkenini kullanmaktadır. Bu değişken sapma sebebini belirtmek için kullanılmıştır. Hedef yol kısıtı π nin bir dallanma koşuludur.

4 Örnek

Bu bölümde örnek olarak verilen üç sayının küçükten büyüğe sıralı olup olmadığını kontrol eden bir program için test kümesi yaratılacaktır. Test altındaki programın kontrol-akış çizelgesi Şekil 2 üzerinde gösterilmiştir.



Şekil 2. sıralıMı(a,b,c) Metodunun Akış Çizelgesi

Klasik bir konkolik test Şekil 3 üzerinde görüldüğü gibi rastgele bir girdi ile başlar. Test altındaki program bu girdiyle çalıştırılır ve tam yol kısıtı π_1 hesaplanır. Bundan sonra yeni bir tam yol kısıtı π_2, π_1 in son dallanma koşulu tersine çevrilerek oluşturulur. Bu yeni çalışma yolunu gerçekleyebilmek için tam yol kısıtı kısıt çözücüye verilir. Bu tam yol kısıtı 3 tane dallanma koşulu içerdiği için bu operasyon 3 uzunlukta kısıt çözümü, kısaca KÇ(3) olarak tanımlanmıştır.

Test altındaki program test boyunca 4 kere çalıştırılmıştır ve kısıt çözücüye 3 kere sorgu gönderilmiştir. Önemli nokta ise bu sorguların ortalama uzunluğunun $(3 + 2 + 1)/3 = 2$ olmasıdır.

Geliştiridiğimiz konkolik test tekniğinde Şekil 4 üzerinde görülen testte bir girdiyle başlanmış ve karşılık gelen tam yol kısıtı öğrenilmiştir. Sonra π_2 normal konkolik testteki gibi oluşturulup hedef olarak belirlenmiştir. Tüm ifadenin çözülmesi yerine tam bu noktada tam yol kısıtının yukarı yaklaşımları kullanılmıştır. ϕ_{21} alttaki kısıt çözücünün programı doğru çalışma yoluna yönlendiremeyen bir girdi yaratmasına neden olmuştur. Bu yüzden algoritma sapmanın nedenini (c_d) bulur ve öğrenir ($\phi \leftarrow \phi \wedge c_d$). Bu öğrenme bir başka yukarı yaklaşım olan $\pi_2 \rightarrow \phi_{22} \rightarrow \phi_{21}$ ile sonuçlanır. Bu yukarı yaklaşım örnekte program çalışmasını hedefe yönlendirme açısından yeterli gelmiştir.

İki algoritmayı karşılaştıracak olursak, bunda da yine test altındaki program 4 kere çalıştırılmıştır, ancak kısıt çözücüye ortalamada $(1 + 2 + 1)/3 = 1.33$ uzunluğunda sorgular yapılmıştır.

$i_1 = [0, 0, 0]$ [başlangıçta]
$P_1 = A \to B \to C \to EVET$
$\pi_1 = (a \leq b) \wedge (a \leq c) \wedge (b \leq c)$
$\pi_2 = (a \leq b) \wedge (a \leq c) \wedge \neg(b \leq c)$
$i_2 = [0, -1, 2]$ [KÇ(3)]
$P_2 = A \to B \to C \to HAYIR$
$\pi_3 = (a \leq b) \wedge \neg(a \leq c)$
$i_3 = [0, 0, -1]$ [KÇ(2)]
$P_3 = A \to B \to HAYIR$
$\pi_4 = \neg(a \leq b)$
$i_4 = [-1, 0, -1]$ [KÇ(1)]
$P_4 = A \to HAYIR$
DUR [Tüm Yollar Denendi]

Şekil 3. Klasik Konkolik Test Kullanılarak Çözüm

$i_1 = [0, 0, 0]$ [başlangıçta]
$P_1 = A \to B \to C \to EVET$
$\pi_1 = (a \leq b) \wedge (a \leq c) \wedge (b \leq c)$
$\pi_2 = (a \leq b) \wedge (a \leq c) \wedge \neg(b \leq c)$
$\phi_{21} = \neg(b \leq c)$
$i_{21} = [0, 0, -1]$ [KÇ(1)]
$P_{21} = A \to B \to HAYIR$ $[P_{21} \neq P_{\text{istenen}}]$
$\phi_{22} = (a \leq c) \wedge \neg(b \leq c)$ $[\phi \leftarrow \phi \wedge c_d]$
$i_{22} = [-1, 0, -1]$ [KÇ(2)]
$P_{22} = A \to B \to C \to HAYIR$ $[P_{22} = P_{\text{istenen}}]$
$\pi_3 = \neg(a \leq b)$
$\phi_{31} = \neg(a \leq b)$
$i_{31} = [0, -1, -1]$ [KÇ(1)]
$P_{31} = A \to HAYIR$
DUR [Tüm Yollar Denendi]

Şekil 4. Geliştiridiğimiz Kısmi Yol Kısıtları ile Çözüm

5 Deneyler

Tekniğimizi CREST adlı C için geliştirilmiş, derinlik-öncelikli arama ve rastgele-dallanma gibi bilinen stratejileri kullanan bir konkolik test uygulaması üzerine ekledik [5]. CREST'i kısmi yol kısıtlarını ve girdi/kısıt önbelleklenmesi kullanacak şekilde değiştirdik. Ayrıca verilen test kümelerinin verilen programlardaki dallanma kapsamını (branch coverage) ölçen bir kod da yazdık.

Aşağıda beş farklı konkolik test uygulaması tanımlanmıştır. Bunlardan KYK ve bKYK bizim geliştiridğimiz kısmi yol kısıtları yaklaşımını kullanmaktayken diğer üçü karşılaştırma amaçlı olarak literatürde bulunana ve CREST'te gerçekleştirlmiş yöntemlerdir.

1. **DÖA:** Derinlik-öncelikli arama kullanan normal konkolik test algoritmasıdır.
2. **KAÇ:** Kontrol-akış çizelgesi yönlendirmeli konkolik test algoritmasıdır.
3. **Rastgele:** Rastgele-dallanma yaklaşımı kullanılan konkolik test algoritmasıdır.
4. **KYK:** Kısmi yol kısıtları yaklaşımı kullanılan konkolik test algoritmasıdır.
5. **bKYK:** Bellekli kısmi yol kısıtları algoritmasıdır. Bu algoritma da normal KYK'dan farklı olarak kısıt çözücüye daha önce yollanmış sorguları ve test altındaki programa gönderilmiş girdileri hatırlayan bir hafıza vardır ve aynı sorgularla çalışmaların tekrarlanmasına engel olmak amacıyla tasarlanmış olup yer kaygıları yüzünden bu makalenin dışında bırakılmıştır.

Beş yöntem de iki farklı C programında (`Dörtgen` ve `Üçgen`) ile çalıştırılmıştır. Bu programlar açıları ve kenar uzunlukları verilen üçgen ve dörtgenleri sınıflandırmak amacıyla yazılmıştır. Test altındaki programın çalıştırılma için iterasyon limiti 100 olarak belirlenmiştir.

Tablo 1'de deney sonuçlarımız verilmiştir. Tablodan KYK ve bKYK'nın DÖA ve KAÇ'a göre daha çok sorgu yaptığı görülebilir. Ancak, ortalama sorgu uzunluğu bunlara göre çok düşüktür. Benzer şekilde Rastgele'nin ortalama sorgu uzunluğunun en kısa olduğu ancak sorgu sayısın çok fazla olduğu ve dallanma kapsamının düştüğü görülmektedir. Bu sayede kısıt çözücünün çözüm süresi kısıt uzunluğuyla üssel olarak arttığından karmaşıklık azaltılmıştır. Tablo'dan bKYK yönteminin kısıt çözücü üzerindeki yükü en az tavizle en çok hafifleten yöntem olduğunu görülmektedir. Önceki sorguları hatırlamanın toplam sorgu sayısı üzerinde bir etkisi olmamıştır, ancak gereksiz test girdilerinin çıkarılması ile test kümeleri KYK'ya göre küçültmüştür. Bu saptama `Üçgen` programının ihtiyaç duyduğu girdi sayısının 27'den 19'a düşürülmüş olmasıyla açıklanabilir.

6 Sonuçlar

Bu ön çalışmada büyüyen kısmi yol kısıtlarının konkolik test için kullanılabilecek bir yaklaşım olduğu gösterilmiştir. Deneylerimizde ortalama sorgu uzunluğunun yarıya kadar düşürülmesi ilerisi için umut vaat edici bir gelişmedir. Bu gelişmelerin özellikle karmaşık yol kısıtlarına sahip programlarda konkolik testin

	Üçgen				Dörtgen			
Yöntem	Sorgu Sayısı	Ort. Sorgu Uzunluğu	Girdi Sayısı	Dallanma Kapsaması	Sorgu Sayısı	Ort. Sorgu Uzunluğu	Girdi Sayısı	Dallanma Kapsaması
DÖA	10	5.3	11	100%	10	5.5	11	100%
KAÇ	19	4.789	19	100%	18	5.06	18	100%
Rastgele	100	2.26	100	85%	100	1.77	100	65%
KYK	26	2.769	27	100%	17	1.5	18	100%
bKYK	26	2.769	19	100%	17	1.5	18	100%

Tablo 1. Üçgen ve Dörtgen Sınıflandırıcılar için Test Sonuçları

daha hızlı çalışmasını sağlayacağı beklenmektedir. Bu öncalışmanın devamı olarak ölçeklenebilirliği görmek açısından daha büyük kodlar üzerinde test etmeyi planlamaktayız.

7 Gelecek Çalışmalar

Kısmi yol kısıtları yaklaşımı birkaç şekilde geliştirilebilir. Örneğin, π için bu kadar küçük bir yukarı yaklaşımla başlamak çok iyi bir fikir değildir. Onun yerine yol üzerinde birbiriyle bağımsız bir dallanma koşulları kümesiyle başlanabilir. Sonra kısıt çözücü bu daha güçlü kısmi kısıtın koşullarını birer birer çözebilir. Bu yöntem kısıt çözücüye yüklenmeden sorgu başına doğru girdileri bulma şansını artırır.

Kısmi yol kısıtları `grep` ve `vim` gibi daha büyük kodlar üzerinde test edilmelidir. Zaman kısıtlarına uyulabilmesi için bu testler henüz gerçekleştirilememiştir.

Kaynaklar

1. P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," *SIGPLAN Not.*, vol. 40, no. 6, pp. 213–223, 2005.
2. A. R. Bradley, "Sat-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011*, pp. 70–87, 2011.
3. C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, 2008.
4. K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 263–272, 2005.
5. J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, 2008.

6. N. Tillmann and J. De Halleux, "Pex: White box test generation for .net," in *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, 2008.
7. K. Sen and G. Agha, "Cute and jcute: Concolic unit testing and explicit path model-checking tools," in *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, 2006.
8. K. Kähkönen, R. Kindermann, K. Heljanko, and I. Niemelä, "Experimental comparison of concolic and random testing for java card applets," in *Model Checking Software* (J. van de Pol and M. Weber, eds.), vol. 6349 of *Lecture Notes in Computer Science*, pp. 22–39, Springer Berlin Heidelberg, 2010.
9. X. Qu and B. Robinson, "A case study of concolic testing tools and their limitations," in *Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement*, ESEM '11, 2011.
10. H. Seo and S. Kim, "How we get there: A context-guided search strategy in concolic testing," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, 2014.
11. S. Sidiroglou-Douskos, E. Lahtinen, N. Rittenhouse, P. Piselli, F. Long, D. Kim, and M. C. Rinard, "Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15*, pp. 473–486, 2015.
12. E. Tsang, "Foundations of constraint satisfaction," 1993.
13. P. Nuzzo, A. Puggelli, S. A. Seshia, and A. Sangiovanni-Vincentelli, "Calcs: Smt solving for non-linear convex constraints," in *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, 2010.
14. M. Souza, M. Borges, M. d'Amorim, and C. S. Pasareanu, "Coral: Solving complex constraints for symbolic pathfinder," in *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, 2011.
15. J. Zhou, M. Yin, and C. Zhou, "New worst-case upper bound for 2-sat and 3-sat with the number of clauses as the parameter," in *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.
16. S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn, "An orchestrated survey of methodologies for automated software test case generation," *J. Syst. Softw.*, vol. 86, pp. 1978–2001, Aug. 2013.
17. B. Dutertre, "Yices 2.2," in *Computer Aided Verification* (A. Biere and R. Bloem, eds.), vol. 8559 of *Lecture Notes in Computer Science*, pp. 737–744, Springer International Publishing, 2014.