ANKARA ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

YÜKSEK LİSANS TEZİ

TEST DURUMU ÖNCELİKLENDİRME

Zafer Can DEMİR

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

ANKARA
2021

ÖZET

Yüksek Lisans Tezi

TEST DURUMU ÖNCELİKLENDİRME

Zafer Can DEMİR

Ankara Üniversitesi
Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı

Danışman: Prof. Dr. Şahin EMRAH

Yazılım yaşam döngüsüne göre bir yazılım geliştirilirken planlama, analiz ve tasarım aşamalarından sonra test aşaması başlar. Test aşamasında yazılımın istenen işlevsellikleri içermesi, içerdiği işlevsellikleri ise doğru bir şekilde yerine getirmesi gibi başlıklar incelenir. Böylece yazılımda hatalı bir kısım var ise düzeltilebilir ve ürün aşamasına gelen yazılım, müşterinin istekleri doğrultusunda teslim edilebilir. Ürün teslimi yapıldıktan sonra ise bakım aşaması başlar. Bu aşamada sonradan fark edilen hatalar, müşteriden gelen geri bildirimler veya yeni bir özellik ekleme gibi sebeplerden dolayı yazılım güncellemesi yapılabilir. Yapılan yazılım güncellemelerinin diğer yazılım işlevselliklerini etkilemediğine emin olmak için ise regresyon testleri kullanılır. Regresyon testleri sürecinde yazılım testi için daha önceden kullanılan testler tekrar koşturulur. Ancak var olan tüm testleri koşturmak zaman ve iş gücü kaybına yol açacağı için maliyetli bir süreçtir. Bu nedenle kullanılan testlere belirli kriterlere göre öncelik değeri vererek ve öncelik değeri yüksek olan testleri daha önce koşarak bir optimizasyon yapılabilir. Böylece gereksiz testlerin yarattığı yük azaltılır ve regresyon testleri amacına daha çabuk ulaşır. Yapılan bu çalışmada bahsedilen probleme dair araştırmalar yapılmış ve problemin çözümü için kullanılan algoritmalar incelenmiştir. Ayrıca hakim küme yöntemi kullanılarak 3 algoritma geliştirilmiş ve bu algoritmaların metasezgisel algoritmalar ile karşılaştırmalı analizleri yapılmıştır.

Mayıs 2021, 55 Sayfa

Anahtar Kelimeler: Test Durumu Önceliklendirme, Regresyon Testleri, Test Durumu Önceliklendirme Algoritmaları, Hakim Küme

ABSTRACT

Master Thesis

TEST CASE PRIORITIZATION

Zafer Can DEMİR

Ankara University
Graduate School of Natural and Applied Sciences
Department of Computer Engineering

Supervisor: Prof. Dr. Şahin EMRAH

When developing a software according to the software lifecycle, the testing phase begins after the planning, analysis and design phases. In the test phase, the topics such as the software to include the desired functionality and the functionality it contains to work correctly are examined. Thus, if there is a faulty part in the software, it can be corrected and delivered according to the customer's wishes as a product. After the delivery of the product, the maintenance phase begins. At this stage, software updates can be made for reasons such as subsequent errors, customer feedback or adding a new feature. Regression tests are used to ensure that software updates do not affect other software functionality. In the process of regression tests, the tests previously used for software testing are run again. However, running all existing tests is a costly process as it will cause time and labor loss. Therefore, an optimization can be made by giving priority to the tests according to certain criteria and running the tests with high priority values first. This reduces the burden of unnecessary tests and regression tests reach their goals more quickly. In this study, researches are made about the problem and algorithms used to solve the problem are examined. In addition, 3 algorithms were introduced using the dominating set method and these algorithms were analyzed comparatively with metaheuristic algorithms.

May 2021, 55 Pages

Keywords: Test Case Prioritization, Regression Tests, Test Case Prioritization Algorithms, Dominating Set

ÖNSÖZ VE TEŞEKKÜR

Söz konusu çalışmalarımda bilgi, yardım ve önerileriyle büyük emeği geçen Ankara Üniversitesi, Fen Bilimleri Enstitüsü, Bilgisayar Mühendisliği Anabilim Dalı öğretim üyesi, Sayın Prof. Dr. Şahin EMRAH'a, araştırmalarımın başında verdiği fikirler ile bana yol gösteren Ankara Üniversitesi, Fen Bilimleri Enstitüsü, Bilgisayar Mühendisliği Anabilim Dalı öğretim üyesi, Sayın Doç. Dr. Mehmet Serdar GÜZEL'e sonsuz şükranlarımı ve saygılarımı sunarım.

Yardımlarını esirgemeyen Ankara Üniversitesi Bilgisayar Mühendisliği Bölümü hocalarıma,

Çalışmalarım sırasında verdiği fikirler ile yanımda olan arkadaşlarım Ece PARLAK ve Can ÜNLÜ'ye,

Son olarak hayatımın her aşamasında yanımda olan ve bana desteklerini esirgemeyen Annem ve Babam'a teşekkürü bir borç bilirim.

Zafer Can DEMİR
Ankara, Mayıs 2021

İÇİNDEKİLER

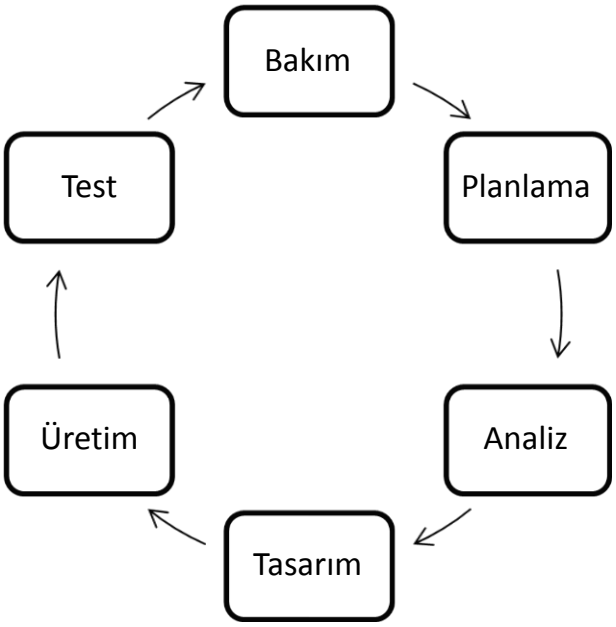
ŞEKİLLER DİZİNİ

ÇİZELGELER DİZİNİ

1. GİRİŞ

Yazılım yaşam döngüsü bir yazılımın geliştirilme ve müşteriye sunulduktan sonraki tüm bakım aşamalarını kapsayan bir süreçtir. Bu süreç 6 aşamadan oluşur ve sırasıyla;

- Planlama aşamasında yazılımın temel ihtiyaçları belirlenir, fizibilite çalışmaları ve görev dağılımı yapılır.
- Analiz aşamasında projenin ne kadar zaman alacağına karar verilir ve olası riskler belirlenir.
- Tasarım aşamasında müşterinin istek ve gereksinimlerine göre ürünün özellikleri belirlenip olası risklerin nasıl çözüleceğine karar verilerek sistem tasarımı ve mimari tasarım süreçleri başlatılır.
- Üretim aşamasında kodlama kısmı yapılır.
- Test aşamasında yazılım olası hatalara karşı analiz edilir.
- Bakım aşamasında ise ürün teslimi yapıldıktan sonra değişen ihtiyaçlar veya bulunan hatalar sonucunda yazılımın farklı bir versiyonunun oluşturulma işlemi gerçekleştirilir (Kılınç 2016, Çetin 2019).



Şekil 1.1 Yazılım yaşam döngüsü

Yazılım yaşam döngüsünün önemli adımlarından birisi test aşamasıdır. Yazılım testi, bir yazılım içerisindeki para, zaman, güven veya can kaybına yol açabilecek, yazılımın normal çalışmasını engelleyen veya hiç çalışmamasına neden olan hataları bulma işlemidir. Günümüzde yazılım gereksinimlerinin farklılaşması ve özelliklerinin artması hatasız yazılım üretilmesini neredeyse imkansız hale getirmiştir. Yazılımda çıkan hatalar müşterinin yazılıma olan ilgisi azaltıp güven kaybına sebebiyet vereceği gibi maliyet ve zaman kaybına da neden olur. Bu hataların kaynakları gereksinim, tasarım, yazılım, yetersiz test veya kodda yapılan diğer güncellemeler olabilir. Hatalar yazılım geliştirme sürecinin ne kadar sonraki bir safhasında fark edilirse bunun maliyeti de o kadar fazla olur. Bu nedenle hataların tespitinin hangi aşamada yapıldığı büyük önem taşır. Örneğin hata gereksinim aşamasında fark edilirse yaratacağı maliyet çok olmazken, tasarım, kodlama ve kullanım aşamalarında katlanarak artar. Yazılım testlerini yapan kişinin amacı hataları mümkün olan en erken aşamada bulmak ve bu hataların giderildiğinden emin olmaktır (Beyhan 2018).

Tarihte yazılım hatalarının büyük kayıpları yol açtığı zamanlar olmuştur. Bunlara örnek olarak;

- Mariner 1: Kağıda yazılan bir formülün kodlama esnasında yanlış yazılması sebebiyle fırlatma sırasında roket yörüngeden çıkmıştır.
- Ariane 5 Flight 501: Ariane 4'ten daha hızlı motora sahip olan bu rokette Ariane 4 kalkış kodu kullanıldığı için kalkıştan 40 saniye sonra parçalara ayrılmıştır.
- Therac-25: Bu radyasyon terapi cihazını kontrol eden yazılımdaki hata sonucu hastalara normalin 125 katından fazla ışın verilmiş ve en az 5 kişi hayatını kaybetmiştir.
- General Electric Energy's XA/21: Yazılımdaki bir yarışma durumundan (race condition) dolayı Amerika ve Kanada'nın belirli bölgelerinde büyük bir güç kesintisi yaşanmıştır.
- Google: Yazılımdaki bir hatadan dolayı arama motoru, kullanıcılara kendi sayfaları da dahil olmak üzere dünya çapındaki tüm web sitelerinin potansiyel olarak zararlı olabileceğine dair bir uyarı vermiştir.

- Mars İklim Aracı: İngiliz ölçü birimlerinin metrik sisteme yanlış çevrilmesi sonucu gezegenler arası iklim aracı Marsa olması gerekenden fazla yaklaşarak imha olmuştur (Beyhan 2018, Anonymous 2019).

Önceki yıllarda yazılım geliştirme sürecinde test aşamasına pek önem verilmezken artık yazılım testi bu döngüde önemli bir yere sahip olmuştur. Bunun ana sebebi müşteriye hata içeren bir ürünü sunmanın maliyetinin gittikçe artmasıdır. Bu maliyetlerden birisi hatanın kaynağını bulmak ve gidermek için gereken iş gücü ve zamandır. Diğer bir maliyet hatanın kullanıcı tarafında şirkete faturalandırılabilecek bir mağduriyet yaratması sonucu oluşan maddi kayıplardır. Bunlar dışında marka imajının olumsuz etkilenmesi sonucu da bir maliyet oluşur (Akdeniz 2017).

1.1 Yazılım Test Metodolojileri

Yazılımlar test edilirken yazılımın tasarımı ve amacına göre farklı yöntemler kullanılır. Bu yöntemler genel olarak iki başlık altında toplanabilir. Bunlar fonksiyonel testler ve fonksiyonel olmayan testlerdir (Luo 2001, Malik vd. 2013).

a) Fonksiyonel Testler

Bu testlere aşağıdaki örnekler verilebilir.

- Birim Testi (Unit Test): Yazılım birimlerinin test edilmesidir. Burada bahsedilen yazılım birimi ise test edilebilen en küçük yazılım bileşenidir ve birim veya modül gibi isimlerle adlandırılırlar. Sınıflar bu birimlere örnek olarak gösterilebilir. Test edilen fonksiyona belirli bir değer gönderilir ve dönen sonucun beklenen sonuç ile aynı olup olmadığına bakılır.
- Bütünleşme Testi (Integration Test): Birim testinin gelişmiş halidir. Birim testinde tek bir fonksiyon test edilirken burada ise birbiri ile bağlantılı fonksiyonların tümü test edilir.
- Arayüz Testi (Interface/UI-User Interface Test): Uygulamaya ait arayüzlerin testleri yapılır. Amaç kullanıcılar için görsellik ve işlevselliğin aynı anda sağlanmasıdır.

- Sistem Testi (System Test): Sadece uygulamanın değil donanım, yazılım ve servisler tümüyle test edilir.
- Regresyon Testi (Regression Test): Yazılımda yapılan bir değişikliğin yazılımın başka bir bölümünü etkileyip etkilemediğinin test edilmesidir.

b) Fonksiyonel Olmayan Testler

Bu testlere aşağıdaki örnekler verilebilir.

- Performans Testi (Performance Test): Uygulama çok data ve çok kullanıcı ile test edilir.
- Yük Testi (Load Test): Uygulama çok kullanıcı ve çok data ile test edildiğinde belirlenen sürede cevap verip veremediği kontrol edilir.
- Stres Testi (Stress Test): Performans ve yük testleri yapıldığında dönen sonuçların doğruluğu kontrol edilir.
- Güvenlik Testi (Security Test): Güvenlik riski taşıyan kısımlar üzerinde yapılan testlerdir.

Bu araştırma kapsamında fonksiyonel testler içinde yer alan regresyon testleri ayrıntılı olarak incelenecektir.

1.2 Regresyon Testleri

Yazılım geliştirme yaşam döngüsüne göre, yazılımın tasarımı yapılıp, implementasyon aşaması tamamlandıktan sonra test aşaması başlar (Li vd. 2007). Bu döngüde test etme ve tekrar test etme süreçleri hataları mümkün olduğunca erken bulabilmek için sürekli tekrar eder (Jeffrey ve Gupta 2006). Bilgi teknolojilerindeki hızlı gelişme ile birlikte bir yazılım kalite güvencesi olarak yazılım testi gün geçtikçe daha önemli bir hale gelmiştir (Jun vd. 2011). Regresyon testi, test aşamasında kullanılan yöntemlerden biridir. Regresyon testlerinde var olan eski testler kullanılırken diğer test yöntemlerinde testler, test eden kişiler tarafından baştan geliştirilir.

Yüksek kalitedeki yazılımlar özenle yapılan test aşamalarından geçmeden geliştirilemez. Modern yazılımların boyutu ve karmaşıklığı arttıkça regresyon testlerinin önemi de artmıştır (Mohapatra ve Prasad 2013). Regresyon testlerinin amacı, düzeltme

veya iyileştirme amaçlı yapılan yazılım değişikliklerinin önceden doğru bir şekilde çalışan diğer yazılım işlevlerini herhangi bir şekilde etkilemediğine emin olmaktır (Ansari vd. 2016). Yazılımda bir değişiklik yapıldığında belli bir test grubu koşturularak bu değişikliğin yazılımın diğer bölümlerini etkilemediğinin kanıtlanması gerekir. Önceden koşturulan testlerin yazılım değişikliği sonrası tekrar koşturularak yapılan değişikliklerin yazılımda yeni hatalar meydana getirmediğine emin olunur. Bu sayede var olan hataları çözerken yeni hatalar oluşturulmamaya çalışılır. Regresyon testleri gereksinim değiştiğinde ve kod da bu doğrultuda güncellendiğinde, yeni bir özellik eklendiğinde ve performans ile ilgili bir iyileştirme yapılacağında gereklidir (Ansari vd. 2016). Regresyon testleri yazılım yaşam döngüsü boyunca tekrarlanır.

Regresyon testleri aşağıdaki 4 durumu kontrol eder (Navdeep vd. 2015).

- Hatalar
- Yazılım güncellemeleri
- Hata düzeltme kusurları
- Performans sorunları

Yukarıda da belirtildiği gibi regresyon testleri hataların bulunması, yazılım güncellemeleri ile yapılmak istenen değişikliklerin doğru yapılması, bir kısım ile ilgili yapılan güncellemenin başka bir kısımda hata oluşturmaması ve güncellemeler ile uygulama performansının düşmemesi konularını kontrol eder.

Regresyon testleri bir yazılım geliştirme sürecidir, yazılım bakımının önemli bir parçasıdır ve önemli bir bütçe gerektirir. Yazılımda yapılan değişikliklerin doğru olduğunu ve yazılımın güncellenmeyen kısımlarının yapılan diğer değişikliklerden etkilenmediğini doğrulamayı sağlar. Bu süreçte yazılım için oluşturulan her testi koşmak çok makul değildir. Çünkü testlerin sayısı arttıkça harcanacak efor ve zaman da artacaktır. Ayrıca tüm testlerin koşturulması için yeterli zaman ve kaynak da olmayabilir. Testleri koştururken oluşacak bu maliyeti azaltmak için bazı teknikler geliştirilmiştir.

Regresyon testleri için aşağıdaki yöntemler kullanılabilir (Srivastava 2008, Singh vd. 2010, Ansari vd. 2016).

1) Tüm testlerin koşturulması: Bu yöntemde yazılım için oluşturulan tüm testlerin yeniden test edilmesi yani bu testlerin tekrar koşulması gerekmektedir. Güncellenmiş yazılıma uygulanamayan testler çıkarılarak, kalan tüm testler tekrar koşulur. Fakat tüm testlerin koşulması fazla zaman alabilir.
2) Regresyon test seçimi: Tüm testlerin tekrar koşulması yerine testlerin belirli bir kısmının seçilmesi şeklinde yapılır. Testler, tekrar kullanılabilenler ve eski, kullanılamayacak testler olarak ayrılır. Sadece tekrar kullanılabilen testlerin sonraki regresyon süreçlerinde kullanılabildiği şekilde test havuzunun bir altkümesi seçilir.
3) Test suit indirgemesi: Yazılıma yeni işlevsellikler eklendikçe gereksiz hale gelen test durumları çıkarılır. Regresyon test seçiminden farklı olarak bu teknikte ihtiyaç duyulmayan testler tamamen kaldırılırken diğer teknikte ise testleri kaldırmak yerine sadece gerekli olan testlerin seçilmesi durumu vardır. Bunun avantajı ise gelecek sürümlerde testlerin doğrulama, koşma ve yönetim maliyetlerini düşürmektir. Dezavantaj olarak ise testlerin hata tespit kapasitesinin düşmesi gösterilebilir.
4) Test durumu önceliklendirme: Belirli amaca uygun, daha önceden belirlenmiş kriterlere göre testlere öncelik değeri verme işlemidir. Bu sayede testleri kullanım dışı bırakmak yerine daha az işe yarayan testler son sıralara atılır. Önceliği yüksek olan testler ise daha önce koşturulur. Bahsedilen kriterler ise hata bulma oranı, maksimum kod kapsama veya önemli özellikleri daha önce kapsama vb. olabilir. Testleri koşmak için kısıtlı bir zaman varsa testlerin koşma süresi de bir kriter olarak eklenebilir.
5) Hibrid Yaklaşım: Test durumu önceliklendirme ve regresyon test seçimi yöntemlerinin beraber kullanılmasıdır.

2. TEST DURUMU ÖNCELİKLENDİRME

Regresyon testleri değişen bir yazılımın düzgün çalıştığını doğrulamak için yapılır. Değişen yazılımın yeni hatalarla gelme ihtimali olduğu ve tüm testleri yeniden koşmak zaman, çaba ve bütçe açısından maliyetli olduğu için testlerin önceliklendirilmesi gerekmektedir (Konsaard ve Ramingwong 2015). Böyle durumlarda test durumu önceliklendirme algoritmaları var olan testleri belirli bir sıraya sokarak regresyon testinin etkisini artırır ve en önemli olanların önce koşulmasını sağlar. Böylece hata tespit işlemi daha erken yapılabilir, test sürecindeki sistemin kalitesi hakkında daha çabuk geri bildirim verilebilir. Zaten regresyon testlerinin amacı da üzerinde değişiklik yapılmış yazılımın daha önceki versiyonu etkileyecek bir hata içerip içermediğini belirlemek ve değişikliklerin hatasız yapıldığını doğrulamaktır (Krishnamoorthi vd. 2009).

Test durumu önceliklendirme algoritmalarının gerekli olduğu alanlardan biri koşum süresidir. Özellikle koşum için belirlenen sürenin kısa olduğu durumlarda bu süre içinde koşulabilecek testlerin belirlenmesi gerekir. Diğer alan ise kod kapsama ile ilgilidir. Yazılım testini yapan kişiler testleri kod kapsamını daha kısa sürede yapacak şekilde sıralamak isteyebilir. Böylece %100 kod kapsamına daha erken ya da belirlenen bir süreden önce ve daha yüksek hızda ulaşılabilir (Krishnamoorthi vd. 2009). Kullanılabilecek bir diğer alan ise gereksinim kapsam oranıdır. Değişen veya müşteriye göre önceliği fazla olan gereksinimlerin daha önce kapsanması gerekmektedir (Kavitha vd. 2010). Testler hata kapsam oranına göre sıralandıysa ilk koşturulacak testin en fazla hatayı kapsadığı anlaşılır (Farooq ve Nadeem 2017). Sıralamadaki kriter toplam durum kapsam önceliği (total statement coverage prioritization) ise, testler kapsadığı durumların sayısına göre azalan şekilde sıraya sokulur. Eklemeli durum kapsam önceliğinde ise, (Additional statement coverage prioritization) testler daha önce kapsanmamış durumları kapsama sayılarına göre sıralanır (Jeffrey ve Gupta 2006). Bazı tekniklerde ise test durumlarının geçmiş koşumlarından elde edilen bilgiler kullanılarak önceliklendirme yapılır (Kim ve Porter 2002). Yeni hatalara sebebiyet veren, birbirine bağımlı ya da diğerlerine göre daha önemli hataları kapsayan test durumlarına öncelik verilerek ise yazılımın çalışmasını en yüksek derecede etkileyen sorunlar giderilebilir.

(Kayes 2011, Wang vd. 2015). Yazılımın bir versiyonunda kalan bir test durumunun yeni bir yazılım versiyonunda kalma olasılığını göz önüne alarak da önceliklendirme yapılabilir (Lin vd. 2013). Bunlar dışında birden fazla kriter göz önüne alınarak hibrid algoritmalarla çözüme ulaşılabilir (Kaur ve Bhatt 2011).

Test durumu önceliklendirme sonucu testler belli kriterlere göre sıralanır. Buradaki amaç ise testlerin sıralanması sonrası, belirlenen amaca sıralanmadan önceki halinden daha kısa sürede ve daha az maliyetle ulaşmaktır (Ansari vd. 2016). Test durumu önceliklendirme teknikleri, var olan testleri, en faydalı olanlar ilk önce koşacak şekilde sıralayarak test sürecinin verimini artırır (Srivastava 2008).

Test durumu önceliklendirme sayesinde;

- Test ekibi tarafında hata bulma oranı artar.
- Yüksek riskli hatalar döngüde daha erken fark edilir.
- Belirli bir kod parçasına bağlı regresyon hataları test sürecinin başlarında fark edilir.
- Kodu kapsama hızı artırılır.
- Sistem daha güvenilir hale gelir (Rothermel vd. 1999).

2.1 Test Durumu Önceliklendirmede Kullanılan Algoritmalar

Test durumu önceliklendirme probleminin çözümünde kullanılan algoritmalardan bazıları aşağıdaki gibidir.

a) Açgözlü Algoritma (Greedy Algorithm)

Açgözlü yaklaşım “bir sonraki en iyi” olanı arama düşüncesine dayanmaktadır. Elemanlara belirli kriterlere göre bir ağırlık değeri verilir ve seçilirken ağırlığı en fazla olan seçilir. Daha sonra ağırlığı en fazla olan ikinci eleman alınır. İstenilen sonuca ulaşana kadar bu işlem tekrarlanır. Örneğin durum kapsamı (statement coverage)

senaryosunu ele alırsak m durum ve n test durumu olduğunu varsayarsak; testlerin hangi durumları kapsadığını bulmanın maliyeti O(mn), test durumlarını quicksort ile sıralama maliyeti O(nlogn), m'nin n'den büyük olduğunu varsayarsak toplam maliyet O(mn) olacaktır (Li vd. 2007).

Açgözlü algoritmalar test durumu önceliklendirmede sıklıkla kullanılır. Bunun nedeni olarak basit ve implementasyonunun kolay olması gösterilebilir. Bu teknik basit olmasına karşın birçok durumda optimal sonuca ulaşamamaktadır. Örneğin Çizelge 2.1'de açgözlü yaklaşım uygulanacak olursa, öncelikle testler kapsadığı durumlara göre sıralanır. En çok durumu kapsayan A (6), sonra B (5) seçilir. C (4) ve D (4) eşit sayıda durum kapsadığı için bulunan sonuçlar A, B, C, D veya A, B, D, C olarak döner. Ancak bu durum için optimal sıralama C, D, A, B veya D, C, A, B olmalıdır.

Çizelge 2.1 Örnek bir durum kapsama tablosu (Li vd. 2007)

Test Case	Statement							
	1	2	3	4	5	6	7	8
A	X	X	X			X	X	X
B	X	X	X				X	X
C	X	X	X	X				
D					X	X	X	X

b) Tamamlayıcı Açgözlü Algoritma (Additional Greedy Algorithm)

Açgözlü algoritma ile benzer olmasına rağmen farklı bir strateji ile çalışır. Bu algoritmada, önceki yapılan seçimler sonucu kapsanmayan durumları kapsayan en fazla ağırlığa sahip eleman seçilir. Bu nedenle her seçim sonucu, kalan elemanların kapsam durumu güncellenir. m durum ve n test durumu olan senaryoda testlerin seçilmesi ve kalan testlerin güncellenmesinin maliyeti O(nm) ve bu işlem O(n) kere tekrarlandığı için toplam maliyet O(mn^2) olacaktır. Çizelge 2.1'deki örnekte 6 durum kapsadığı için öncelikle A seçilir. Geriye 4 ve 5 numaralı durumlar kapsanmamış olarak kalır. Bu durumda B seçilmez, çünkü bu test 4 ve 5 numaralı durumlardan herhangi birini

kapsamaz. C ve D bu iki durumu da kapsadığı için bu algoritmanın döneceği sonuç ya A, C, D, B ya da A, D, C, B olacaktır (Li vd. 2007).

c) 2-Optimal Algoritma

Bu algoritma K-Optimal Açgözlü yaklaşımın bir varyasyonudur. Bu algoritmada problemin en büyük kısmını çözen K eleman birlikte alınır. K-Optimal Tamamlayıcı Algoritmada (K-Optimal Additional Greedy) ise problemin kalan kısmını çözen K eleman alınır. 2-Optimal Algoritmada K=2 olarak seçilmiştir. m durum ve n test durumu olan senaryoda test ikililerinin seçilmesi ve kalan testlerin kapsam durumlarının güncellenmesi işlemi O(mn^2) ve bu işlem O(n) kere tekrarlandığı için toplam maliyet O(mn^3) olacaktır. Çizelge 2.1'de toplamları 8 durum kapsadığı için C ve D öncelikli olarak seçilir. Algoritma ise C, D, A, B sonucunu döner (Li vd. 2007).

d) Genetik Algoritma

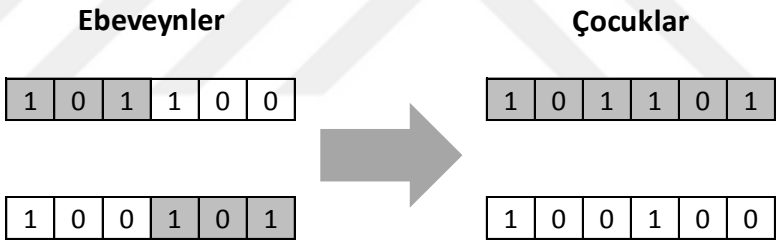
Darwin'in biyolojik evrim teorisine dayanan bir doğal genetik seleksiyon algoritmasıdır. En iyi olan hayatta kalır fikriyle çalışır. Hücrelerdeki kromozom ve kromozomlardaki DNA baz alınarak bir model oluşturulur. Bu algoritmadaki en küçük birim gendir. Genler kromozomları (bireyleri), kromozomlar popülasyonu oluşturur. Öncelikle çözüm tahmin edilir. Daha sonra en iyi çözümleri kullanıp yeni jenerasyonlar oluşturularak problem çözülmeye çalışılır.

Başlangıç popülasyonu oluşturma, bireyleri değerlendirme, ağırlığı fazla olan bireylerden bir çift seçip bunlardan yeni birey oluşturma, bireylerin mutasyona uğraması ve yeni jenerasyonun oluşturulması gibi evreler vardır.

Başlangıç popülasyonu rastgele üretilen bireyler tarafından oluşturulur. Her birey (kromozom), gen denilen bir dizi değişken ile temsil edilir. Kromozomlar belirlenen problemler için olası bir çözüm değeri taşır. Bu kromozomların bir araya gelmesi ile de popülasyon oluşur. Bireyin popülasyondaki değeri oluşturulan uygunluk fonksiyonu

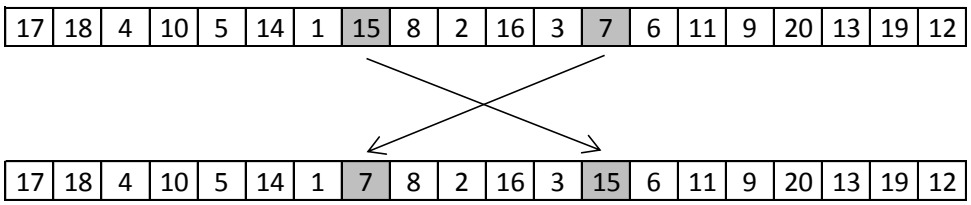
(fitness function) aracılığıyla belirlenir. Bu fonksiyondan çıkan değerler sonucunda en fazla uyuma sahip bireyler genetik değişim (crossover) denilen işlemden geçerek yeni jenerasyonları oluştururlar. Bu işlem basit tabirle iki bireyin genlerini değiştirme işlemidir. Mutasyon işleminde ise bireyin sahip olduğu gen veya genler değişime uğrayarak genetik değişkenlik sağlamaktadır. Bulunan sonuçlar istenilen yeterliliğe sahipse veya belirlenen jenerasyon sayısına ulaşıldıysa işlemler bitmiş olur.

Genetik algoritma test durumu önceliklendirme problemine uygulanırken her test bir gen, test durumlarının bir permütasyonu ise bir kromozom olarak kullanılmıştır. Genetik değişim işleminde rastgele bir k noktası belirlenir. Birey 1'in ilk k noktası Çocuk1'in ilk k elemanını, Birey2'nin k noktasından sonraki elemanları Çocuk1'in k noktasından sonraki elemanlarını oluşturur. Aynı şekilde Birey2'nin ilk k noktası Çocuk2'in ilk k elemanını, Birey1'nin k noktasından sonraki elemanları Çocuk2'in k noktasından sonraki elemanlarını oluşturur.



Şekil 2.1 Genetik algoritmada genetik değişim

Mutasyon işleminde rastgele iki gen seçilerek bunlar yer değiştirilir.



Şekil 2.2 Genetik algoritmada mutasyon

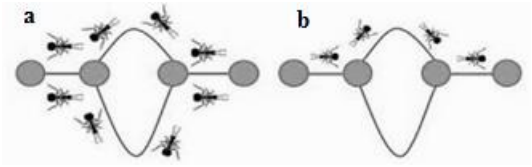
Bunlar dışında popülasyon boyutu, mutasyon olasılığı, genetik değişim olasılığı, jenerasyon sayısı gibi parametreler belirlenmelidir.

e) Karınca Koloni Algoritması (Ant Colony Optimization)

Bu yöntem karıncaların yiyecek ararken yaptıkları davranışlar incelenerek geliştirilmiştir. Bir karınca yiyecek ararken gezindiği yola feromon denilen bir madde bırakır. Bu madde iki amaç için kullanılır. Birincisi diğer gezinen karıncaların bu yolu takip etmesini sağlar. İkincisi ise yolu belirleyen karıncanın yuvaya dönüşünü sağlar. Feromon nedeniyle bu yolu takip eden karıncalar da pozitif bir geri dönüş olarak kendi salgıladıkları feromon ile bu maddenin yoldaki yoğunluğunu artıracaklardır. Yeni bir karınca geldiğinde ise feromon miktarı fazla olan yolu seçecektir. Zamanla buharlaşan feromon ise lokal optimum probleminden kaçınmayı sağlar. En kısa yol üzerinden yiyeceğe ulaşan bir karınca aynı zamanda yuvaya da en önce dönecektir. Aynı yolda yiyeceğe gidip gelerek ise bu yolda feromon bırakarak en optimal çözümün bu yol olduğunu belirtecektir. Deneme yanılma ile farklı yollar deneyen diğer karıncalar ise zamanla bu yolun en kısa yol olduğunu anlayacaklardır. Karıncaların yuvalarında yiyeceğe uzanan bu yoldaki bu davranışına ise karınca koloni optimizasyonu adı verilir (Singh vd. 2010).

Bu algoritmada öncelikle testler hata kapsamlarına göre sıralanır. En çok kapsama sahip test seçilir. Daha sonra kalan hatalar belirlenerek bu hataların en çoğunu kapsayan test belirlenerek o test seçilir. Aynı şekilde tüm hatalar kapsanana kadar test seçimi devam eder. Daha sonra her bir testin kapsadığı hata sayısı belirlenir. Bu testlerin kombinasyonu ise yol (path) denen ve tüm hataların kapsandığı bir çözüm üretecektir (Ansari vd. 2016).

Algoritmada testlerin kapsadığı hata sayısı, testlerin koşma süresi, feromon matrisi girdi olarak verilir. Feromon değeri kapsanan hata / koşma süresi ile hesaplanır. En çok hatayı en az sürede kapsayan ve feromon değeri en yüksek olan yol seçilir (Ansari vd. 2016).



Şekil 2.3 (a) Karıncaların rastgele izledikleri yol (b) Karıncaların belirlediği en kısa yol (Ansari vd. 2016)

f) Parçacık Sürü Optimizasyonu (Particle Swarm Optimization)

Bu algoritma sürü halinde hareket eden kuş ve arı gibi hayvanların hareketleri göz önüne alınarak oluşturulmuştur. Bu hayvanların doğal ortamlarındaki hareketleri incelendiğinde yiyecek bulma gibi en temel ihtiyaçlarını karşılarken bazı hareketler sergiledikleri görülmüştür. Bu hareketlerin diğer sürü bireyleri ile etkileşimli olarak yapıldığı ve sürünün belirlediği amaca daha hızlı ve kolay ulaşabildiği gözlemlenmiştir. Parçacık sürü optimizasyonu algoritmasında bireyler tek başlarına yapamadıkları işleri, diğer sürü üyeleriyle birlikte hareket ederek, onların da bilgilerini kullanarak yaparlar. Bilgi paylaşımı ile grup bilgisi optimize edilir ve edinilen ortak bilgi ile toplu yönelim yapılır (Kennedy ve Eberhart 1995).

Şekil 2.4 Sürüdeki kuşların birlikte hareketi

Parçacık sürü optimizasyonunda her bireye parçacık, bu parçacıkların oluşturduğu popülasyona ise sürü adı verilir. Her bireyin belirlenen hedefe ne kadar yaklaştığını hesaplamak için uygunluk fonksiyonu kullanılır. Bir parçacığın çözüme en fazla yaklaştığı duruma pbest, tüm sürüdeki parçacıklardan çözüme en çok yaklaşanın durumuna ise gbest denir. Bu değerler kullanılarak sonlandırma kriterine uygunluğu göz önüne alınarak algoritma sonlandırılır (Hla vd. 2008).

Algoritmanın işleyişi şu şekildedir;

- Kullanılacak parametreler ve sürünün ilk pozisyonu belirlenir.
- Sürüdeki parçacıkların uygunluk değeri hesaplanır.
- gbest ve pbest değerleri güncellenir.
- Parçacıkların değişim hızları hesaplanıp yeni pozisyonları belirlenir.
- Sonlandırma kriterine ulaşıldıysa algoritma sonlandırılır, ulaşılmadıysa aynı işlemler tekrarlanır.

g) K-Ortalamalar Kümeleme (K-Means Clustering)

Bu yöntemde amaç, birbirine benzer olan test durumlarını tespit etmek üzerinedir. Test durumlarını karşılaştırmak için önceden belirlenen özellikler kullanılarak, havuzdaki test durumlarının birbirleriyle olan benzerlikleri tespit edilir. Bu belirlenen özellikler bir matris yardımıyla analiz edilir ve birbirlerine benzeyen özellikler aşırı uyum gösterme (overfitting) problemine neden olmaması için çıkarılır. Küme sayısının belirlenmesinden sonra ise test durumları kümelenmeye başlanır. Benzer özellikte olan test durumları aynı kümede olacak şekilde, belirlenen sayıda küme elde edilir. Her kümede önceliklendirme teknikleri uygulanarak bir sıralama yapılır. Bu yapılan sıralama sonucu ise bazı test durumları önceliklendirilmiş olur (Gokilavani vd. 2021).

2.2 Algoritmaların Karşılaştırılması

Yukarıda belirtilen algoritmalar karşılaştırıldığında, küçük sayıdaki test durumları ile yapılan denemelerde algoritmalar benzer performanslar göstermiştir. Sayı arttıkça ise performans farkları oluşmaya başlamıştır. Açgözlü algoritmalar deterministik olduğu için birkaç kere denendiğinde sürekli aynı sonucu verecektir. Ancak genetik algoritma vb. diğer algoritmalarda ise farklı sonuçlar verecektir. Belirlenen sonuçlara göre genetik algoritmanın diğer algoritmalara göre daha iyi performans sergilediği söylenebilir (Li vd. 2007).

3. HAKİM KÜME YÖNTEMİYLE TEST DURUMU ÖNCELİKLENDİRME

3.1 Hakim Küme (Dominating Set)

Verilen bir G(V,E) çizgesi içerisindeki düğümlerin, boş olmayan bir alt kümesi olan D için, çizgedeki her düğüm ya bu altküme içerisinde ise ya da bu altküme içerisindeki herhangi bir düğüme komşu ise, D'ye hakim küme denir. Eleman sayısı en az olan hakim kümeye ise minimum hakim küme denir. Bu problem NP-Zor'dur ve çözümü için birçok algoritma geliştirilmiştir (Liedlof 2008).

Bu tez kapsamında testlerin çalışma süresinin eşit olduğu varsayılıp, testlerin izlenebilirlik verdiği (kapsadığı) gereksinim sayıları üzerinden işlemler yapılmıştır. Bu doğrultuda tanımlanan, testler kümesi X ve gereksinimler kümesi Y'de, seçilen herhangi bir x_1 testi için, bu testin izlenebilirlik verdiği y_1, y_2, y_3 ... gereksinimlerine bir ok çizilirse, X kümesinden Y kümesine iki kümeli yönlü bir çizge (directed bipartite graph) oluşacaktır. Test durumu önceliklendirme problemini çözmek için ise oluşan bu çizgede minimum hakim kümeyi bulmamız yeterli olacaktır.

Çizge yöntemi ile test durumu önceliklendirme probleminin çözümü üzerine daha önce çeşitli çalışmalar yapılmıştır. Örneğin bir çalışmada kod kapsama, test koşum süresi ve test benzerliği kullanılarak test durumlarından oluşan bir çizge elde edilmiştir. Kod kapsama ve test koşum süresi düğüm değerini etkilerken test benzerliği düğümler arasındaki kenarların değerlerini belirlenmiştir. Daha sonra ise çizge üzerinde gezinerek test durumları önceliklendirilmiştir (Azizi ve Do 2018). Başka bir çalışmada koda yapılan enstrümantasyon işleminden sonra yapılan metot çağrıları bir çizge formatına dönüştürülmüştür. Daha sonra elde edilen çizge, çağrı çizgesi adı altında sadeleştirilmiştir. Düğümler arasındaki kenarlara, yapılan metot çağrılarının türüne göre (test-test, test-kaynak kod ve kaynak kod-kaynak kod) farklı değerler atanarak üzerinde işlemler yapılmıştır (Chi vd. 2020). Bir diğer çalışmada ise testler benzeşmezlik (dissimilarity) oranları kullanılarak çizgeye aktarılmıştır. Daha sonra ise bu çizgede farklı sıralama algoritmaları kullanılarak problem çözülmeye çalışılmıştır (Murali vd. 2008). Bu tez kapsamında ise testler ve gereksinimler arasındaki ilişkiler kullanılarak problem çözülmeye çalışılmıştır. Gereksinimler aslında belli bir kaynak kod parçasının başlığı olarak düşünülebilir. Test durumu içinde izlenebilirlik verilen gereksinimlerden

yola çıkılarak, ilgili testin kaynak kodun hangi kısmını kapsayacağı anlaşılabilir. Bu mantıkla testler ve gereksinimler çizgedeki düğümler olacak şekilde, testler ve bu testlerin kapsadığı gereksinimler arasına yönlü bir bağlantı eklenerek ilişki kurulmuştur. Daha sonra ise bu çizgeye hakim küme yöntemi uygulanarak problem çözülmeye çalışılmıştır.

3.2 Geliştirilen Algoritmalar

Bu tez kapsamında hakim küme yöntemiyle test durumu önceliklendirme yapabilmek için 3 adet algoritma geliştirilmiştir. Bu algoritmalar şunlardır:

- H1 - LessCoveredReqFirst
- H2 - MostCoveringTestCaseFirst
- H3 - MostCoveringTestCaseFirst_Dyn

Algoritma açıklamalarında aşağıdaki kısaltmalar kullanılacaktır.

- Test durumları havuzu: T
- Gereksinimler: R
- Kapsanması gereken gereksinimler: r

3.2.1 H1- LessCoveredReqFirst

1. T içindeki her bir t için, r'den herhangi bir elemanı kapsayanlar belirlenir. (Çizgenin sol tarafındaki testlerden, sağ tarafta belirlenen altküme gereksinimler (r) arasında kenar olanlar belirlenir)
2. Bu test durumlarının, t_n, r içindeki kapsadığı gereksinimlerin listesi, r_n , belirlenir.
3. Bu test durumları "Key = t_n ID", "Value = r_n" formatında bir dictionary D_1 içinde tutulur. (Dictionary<int, List<int>>)
4. D_1 kullanılarak r içindeki her bir gereksinimin testler tarafından kaç kez kapsandığı (n) belirlenir (Gereksinimin, çizgenin sol tarafındaki testlerle sahip olduğu kenar sayısı belirlenir) ve "Key = R_n ID", "Value = n" olarak dictionary D_2 içinde tutulur. (Dictionary<int, int>)

5. Son durumdaki test durumlarının ID'lerini tutacak bir F listesi oluşturulur. (List<int>)
6. D_2, n sayısına göre küçükten büyüğe göre sıralanır.
7. D_2'nin ilk elemanı (gereksinim) seçilir, "Value = ∞" ise program sonlandırılır ve Adım 12'ye gidilir.
8. D_1 içinde, D_2'nin ilk elemanını kapsayan ilk test durumu seçilir (Çizgede gereksinim ile arasında kenar olan ilk test seçilir) ve F içine bu test durumunun ID'si eklenir.
9. D_1 içindeki her bir elemanın Value listesinden, seçilen test durumunun Value listesi çıkarılarak güncellenir. Böylece kalan gereksinimleri kapsayan testler belirlenmiş olur. (Graphta kapsanan gereksinim ve bu gereksinimle kenarı olan testlerin arasındaki bağlantılar silinir.)
10. Seçilen test durumunun Value listesinde bulunan her gereksinim kullanılarak, o gereksinim D_2 içinde "Value = ∞" olacak şekilde güncellenir. Böylece kapsanan gereksinimler çıkarılmış olur.
11. Adım 6'ya gidilir.
12. F içindeki testler, seçilen testleri oluşturmaktadır.

3.2.2 H2 - MostCoveringTestCaseFirst

1. T içindeki her bir t için, r'den herhangi bir elemanı kapsayanlar belirlenir. (Çizgenin sol tarafındaki testlerden, sağ tarafta belirlenen altküme gereksinimler (r) arasında kenar olanlar belirlenir)
2. Bu test durumlarının, t_n, r içindeki kapsadığı gereksinimlerin listesi, r_n , belirlenir.
3. Bu test durumları "Key = t_n ID", "Value = r_n" formatında bir dictionary D_1 içinde tutulur. (Dictionary<int, List<int>>)
4. Son durumdaki test durumlarının ID'lerini tutacak bir F listesi oluşturulur. (List<int>)
5. D_1, Value listesindeki elemanların sayılarına göre büyükten küçüğe sıralanır.
6. D_1'in ilk elemanı seçilir, "Value Count = 0" ise program sonlandırılır ve Adım 10'a gidilir.
7. F içine bu test durumunun ID'si eklenir.

8. D_1 içindeki her bir elemanın Value listesinden, seçilen test durumunun Value listesi çıkarılarak güncellenir. Böylece kalan gereksinimleri kapsayan testler belirlenmiş olur. (Graphta kapsanan gereksinim ve bu gereksinimle kenarı olan testlerin arasındaki bağlantılar silinir.)
9. Adım 5'e gidilir.
10. F içindeki testler, seçilen testleri oluşturmaktadır.

3.2.3 H3 - MostCoveringTestCaseFirst_Dyn

1. T içindeki her bir t için, r'den herhangi bir elemanı kapsayanlar belirlenir. (Çizgenin sol tarafındaki testlerden, sağ tarafta belirlenen altküme gereksinimler (r) arasında kenar olanlar belirlenir)
2. Bu test durumlarının, t_n, r içindeki kapsadığı gereksinimlerin listesi, r_n , belirlenir.
3. Bu test durumları "Key = t_n ID", "Value = r_n" formatında bir dictionary D_1 içinde tutulur. (Dictionary<int, List<int>>)
4. Son durumdaki test durumlarının ID'lerini tutacak bir F listesi oluşturulur. (List<int>)
5. Local olarak test durumlarının ID'lerini tutacak bir f listesi oluşturulur. (List<int>)
6. Son durumdaki test durumlarının sayısını tutacak "GlobalCount = ∞" tanımlanır.
7. Method parametreleri $P_1 = D_1$ ve P_2 = f olarak belirlenir.
8. SelectTestCases_TestCase_Dynamic methodu belirlenen parametrelerle çağırılır.
9. P_1, Value listesindeki elemanların sayılarına göre büyükten küçüğe sıralanır.
10. Value listesindeki elemanların sayıları eşit olan testleri tutacak "maxCoveringElements" (Dictionary<int, List<int>>) tanımlanır.
11. P_1'in ilk elemanı seçilir, "Value Count = 0" ise ve P_2.Count (P_2 listesinin eleman sayısı) GlobalCount'tan küçükse "F=P_2" ve "GlobalCount = P_2.Count" olarak güncellenir. (Daha iyi bir çözüm bulunduğunda güncelleme yapılır.)
12. "Value Count = 1" ise maxCoveringElements'e bu eleman eklenir. (Test 1 gereksinim kapsıyorsa, daha iyi bir çözüm bulunamayacağı için direk seçilir.)
13. "Value Count = x" ise (x, 0 ve 1'den farklı bir sayı), P_1 içinde "Value Count = x" olan ve seçilen elemanın kapsadığı gereksinim listesinden farklı en az 1 gereksinim kapsayan tüm elemanlar maxCoveringElements'e eklenir. Böylece aynı sayıda ama

farklı gereksinimleri kapsayan tüm testler seçilmiş olur. (Çizgede gereksinimlerle bağlantı sayıları eşit olan testler tespit edilir.)

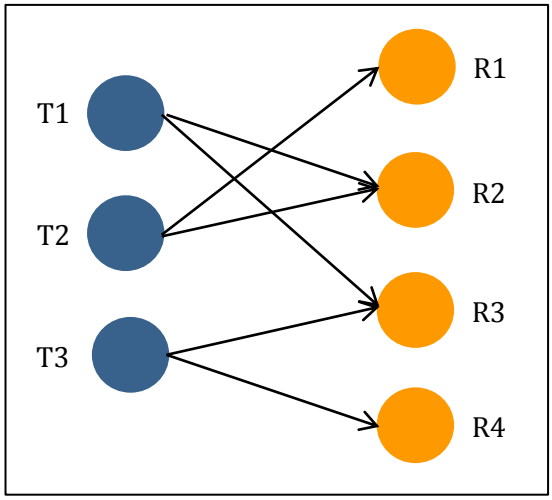
14. maxCoveringElements'teki elemanların her biri için P_1 ve P_2 in kopyaları D_{1_dup} ve f_{dup} oluşturulur.
15. f_{dup} içine seçilen eleman eklenir.
16. D_{1_dup} içindeki her bir elemanın Value listesinden, seçilen test durumunun Value listesi çıkarılarak güncellenir. Böylece kalan gereksinimleri kapsayan testler belirlenmiş olur. (Graphta kapsanan gereksinim ve bu gereksinimle kenarı olan testlerin arasındaki bağlantılar silinir.)
17. Method parametreleri $P_1 = D_{1\_dup}$ ve $P_2 = f_{dup}$ olarak belirlenir.
18. Adım 8'e gidilir.

3.2.4 Algoritmaların örnek üzerinde açıklanması

Çizelge 3.1'de, algoritmaların uygulanacağı örnek bir test ve gereksinim tablosu verilmiştir. Bu tabloda her testin kapsadığı gereksinim işaretlenerek belirtilmiştir. Bu tablo ayrıca Şekil 3.1'de çizge üzerinde gösterilmiştir.

Çizelge 3.1 Örnek test ve gereksinim tablosu

	R1	R2	R3	R4
T1		x	x	
T2	x	x		
T3			x	x



Şekil 3.1 Örnek test ve gereksinim tablosunun çizge üzerinde gösterimi

a) H1 - LessCoveredReqFirst Algoritması

- Her gereksinimin testler tarafından kapsanma sayısı belirlenir.

Çizelge 3.2 H1 algoritması gereksinim kapsanma sayıları

Gereksinim	Kapsanma Sayısı
R1	1
R2	2
R3	2
R4	1

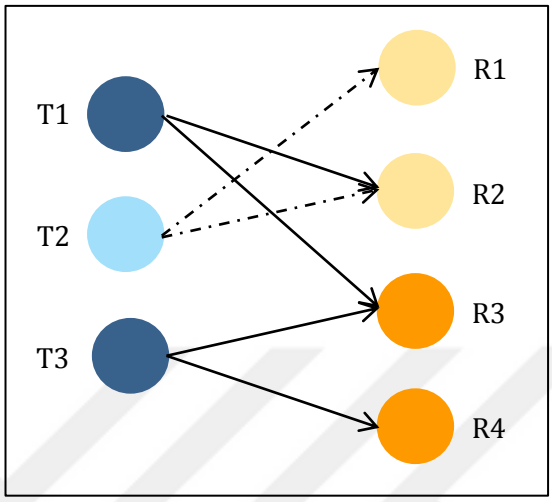
- Gereksinimler kapsanma sayılarına göre küçükten büyüğe doğru sıralanır.

Çizelge 3.3 H1 algoritması sıralanmış gereksinim kapsanma sayıları

Gereksinim	Kapsanma Sayısı
R1	1
R4	1
R2	2
R3	2

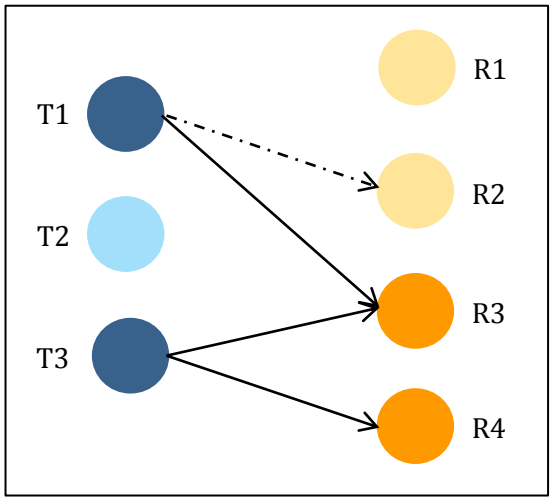
- İlk eleman olan R1'i kapsayan ilk test seçilir. (T2)

- T2 ve kapsadığı tüm gereksinimler arasındaki bağlantılar silinerek, bu gereksinimler, kapsanan gereksinimler arasına eklenir. T2, seçilen testlere eklenir.



Şekil 3.2 H1 algoritması T2'nin seçilmesi

- Kapsanan gereksinimler (R1 ve R2) ve bu gereksinimlere izlenebilirlik veren diğer testler arasındaki bağlantılar silinir.



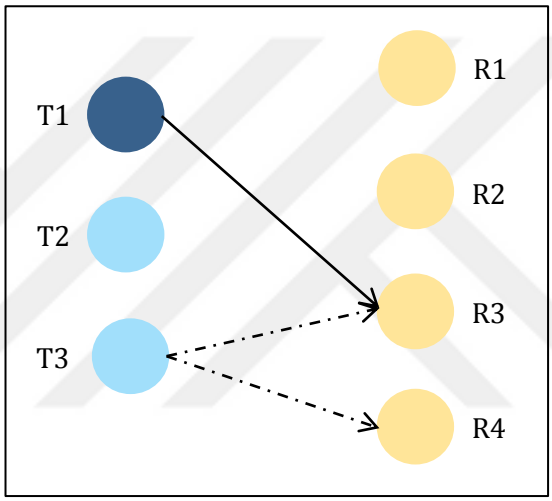
Şekil 3.3 H1 algoritması R1 ve R2'yi kapsayan testlerin bağlantılarının silinmesi

- Kalan gereksinimlerin kapsanma sayıları belirlenir ve küçükten büyüğe sıralanır.

Çizelge 3.4 H1 algoritması kalan gereksinimlerin kapsanma sayıları

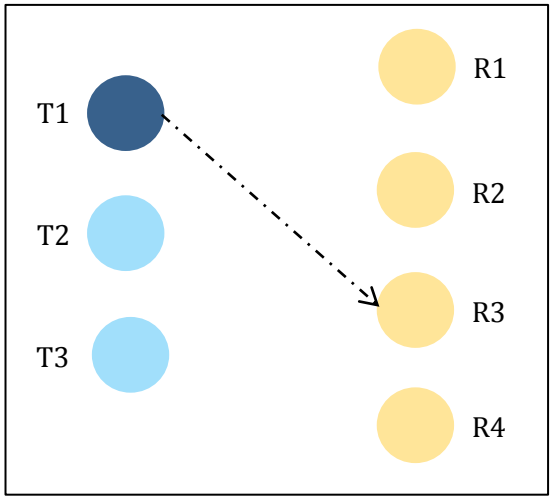
Gereksinim	Kapsanma Sayısı
R4	1
R3	2

- İlk eleman olan R4'ü kapsayan ilk test seçilir. (T3)
- T3 ve kapsadığı tüm gereksinimler arasındaki bağlantılar silinerek, bu gereksinimler, kapsanan gereksinimler arasına eklenir. T3, seçilen testlere eklenir.



Şekil 3.4 H1 algoritması T3'ün seçilmesi

- Kapsanan gereksinimler (R3 ve R4) ve bu gereksinimlere izlenebilirlik veren diğer testler arasındaki bağlantılar silinir.



Şekil 3.5 H1 algoritması R3 ve R4'ü kapsayan testlerin bağlantılarının silinmesi

- Tüm gereksinimler kapsandığı için program sonlanır.
- Seçilen Testler: T2, T3

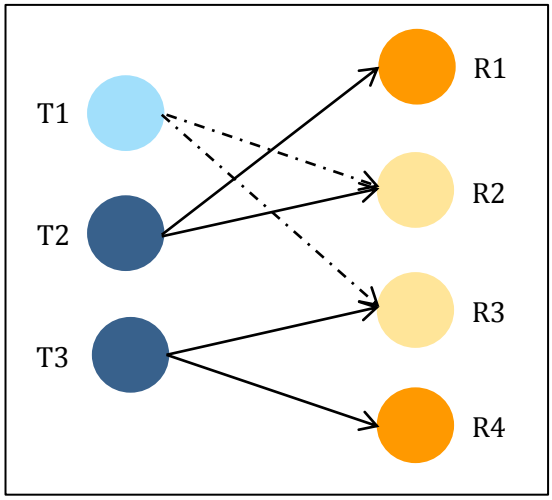
b) H2 - MostCoveringTestCaseFirst Algoritması

- Testler gereksinim kapsama sayılarına göre büyükten küçüğe göre sıralanır.

Çizelge 3.5 H2 algoritması testlerin gereksinim kapsama sayıları

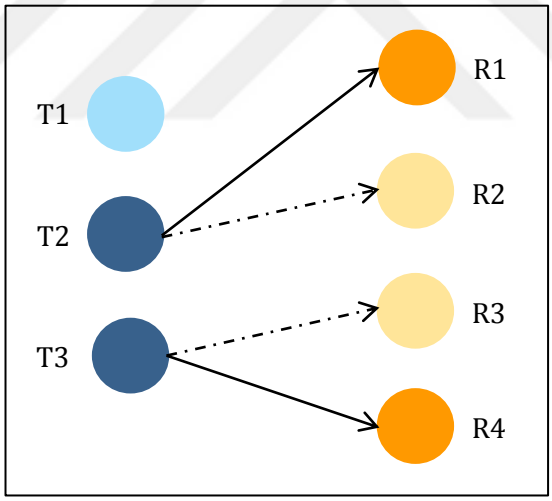
Test	Kapsama Sayısı
T1	2
T2	2
T3	2

- İlk eleman olan T1 seçilir.
- T1 ve kapsadığı tüm gereksinimler arasındaki bağlantılar silinerek, bu gereksinimler, kapsanan gereksinimler arasına eklenir. T1, seçilen testlere eklenir.



Şekil 3.6 H2 algoritması T1'in seçilmesi

- Kapsanan gereksinimler (R2 ve R3) ve bu gereksinimlere izlenebilirlik veren diğer testler arasındaki bağlantılar silinir.



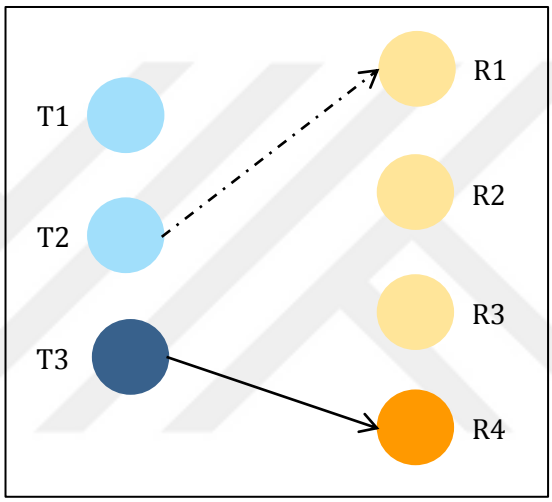
Şekil 3.7 H2 algoritması R2 ve R3'ü kapsayan testlerin bağlantılarının silinmesi

- Kalan gereksinimler üzerinden (R1 ve R4), testlerin kapsama sayıları büyükten küçüğe sıralanır.

Çizelge 3.6 H2 algoritması kalan testlerin gereksinim kapsama sayıları

Test	Kapsama Sayısı
T2	1
T3	1

- İlk eleman olan T2 seçilir.
- T2 ve kapsadığı tüm gereksinimler arasındaki bağlantılar silinerek, bu gereksinimler, kapsanan gereksinimler arasına eklenir. T2, seçilen testlere eklenir.



Şekil 3.8 H2 algoritması T2'nin seçilmesi

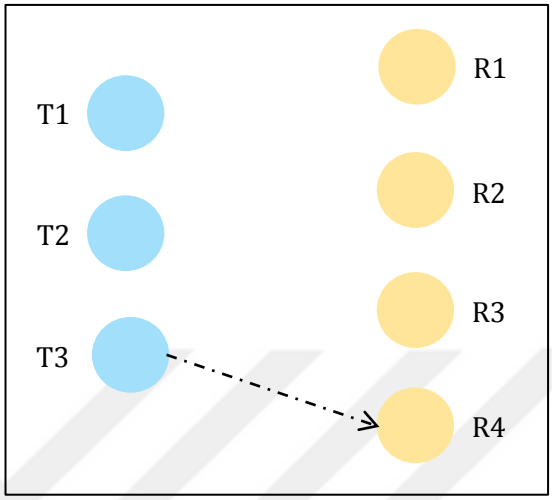
- Kapsanan gereksinimler (R1) ve bu gereksinimlere izlenebilirlik veren diğer testler arasındaki bağlantılar silinir. R1 ile T2 hariç herhangi bir bağlantı olmadığı için sonraki adıma geçilir.
- Kalan gereksinimler üzerinden (R4), testlerin kapsama sayıları büyükten küçüğe sıralanır.

Çizelge 3.7 H2 algoritması son kalan testin gereksinim kapsama sayısı

Test	Kapsama Sayısı
T3	1

- İlk eleman olan T3 seçilir.

- T3 ve kapsadığı tüm gereksinimler arasındaki bağlantılar silinerek, bu gereksinimler, kapsanan gereksinimler arasına eklenir. T3, seçilen testlere eklenir.



Şekil 3.9 H2 algoritması T3'ün seçilmesi

- Kapsanan gereksinimler (R4) ve bu gereksinimlere izlenebilirlik veren diğer testler arasındaki bağlantılar silinir. R4 ile T3 hariç herhangi bir bağlantı olmadığı için sonraki adıma geçilir.
- Tüm gereksinimler kapsandığı için program sonlanır.
- Seçilen Testler: T1, T2, T3

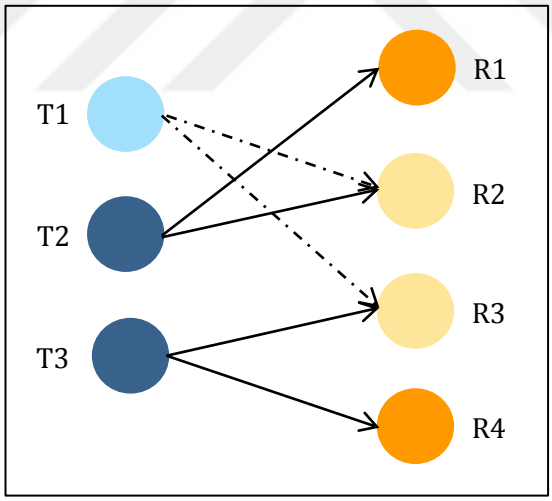
c) H3 - MostCoveringTestCaseFirst_Dyn

- F (Son durumdaki testleri tutacak olan liste), GlobalCount = ∞ (Son durumdaki seçilecek test sayılarını tutacak değişken) ve maxCoveringElements listesi (ilk eleman ile aynı sayıda gereksinim kapsayan testleri tutacak liste) tanımlanır.
- Testler gereksinim kapsama sayılarına göre büyükten küçüğe göre sıralanır.

Çizelge 3.8 H3 algoritması testlerin gereksinim kapsama sayıları

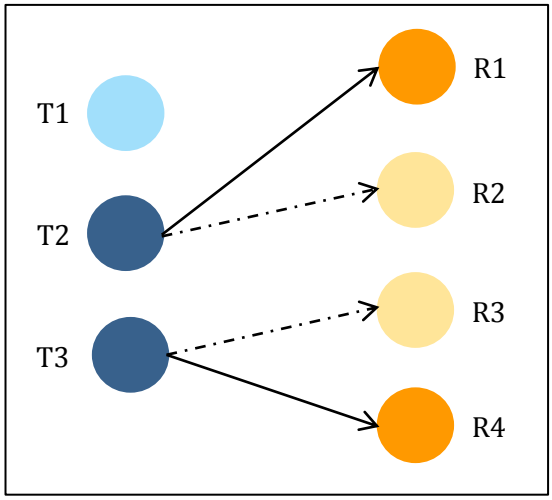
Test	Kapsama Sayısı
T1	2
T2	2
T3	2

- İlk eleman olan T1 seçilir.
- T1 ile aynı kapsama sayısına sahip testler belirlenir ve bunlar T1 ile beraber maxCoveringElements listesine eklenir. (maxCoveringElements içinde T1, T2 ve T3 oldu)
- Bu listedeki her bir eleman için aşağıdaki işlemler yapılır.
- **İterasyon 1 (T1 için):** T1 seçilir.
- **İterasyon 1 (T1 için):** T1 ve kapsadığı tüm gereksinimler arasındaki bağlantılar silinerek, bu gereksinimler, kapsanan gereksinimler arasına eklenir. T1, seçilen testlere eklenir.



Şekil 3.10 H3 algoritması - İterasyon 1, T1'in seçilmesi

- **İterasyon 1 (T1 için):** Kapsanan gereksinimler (R2 ve R3) ve bu gereksinimlere izlenebilirlik veren diğer testler arasındaki bağlantılar silinir.



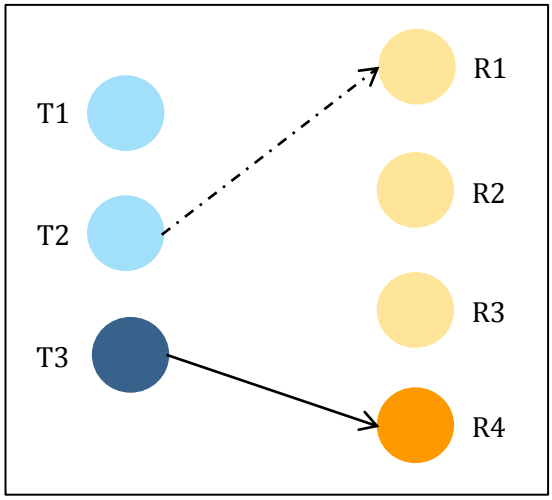
Şekil 3.11 H3 İterasyon 1, R2 ve R3'ü kapsayan testlerin bağlantılarının silinmesi

- **İterasyon 1 (T1 için):** Kalan gereksinimler üzerinden (R1 ve R4), testlerin kapsama sayıları büyükten küçüğe sıralanır.

Çizelge 3.9 H3 İterasyon 1, kalan testlerin gereksinim kapsama sayıları

Test	Kapsama Sayısı
T2	1
T3	1

- **İterasyon 1 (T1 için):** İlk eleman olan T2 seçilir.
- **İterasyon 1 (T1 için):** T2 ve kapsadığı tüm gereksinimler arasındaki bağlantılar silinerek, bu gereksinimler, kapsanan gereksinimler arasına eklenir. T2, seçilen testlere eklenir.



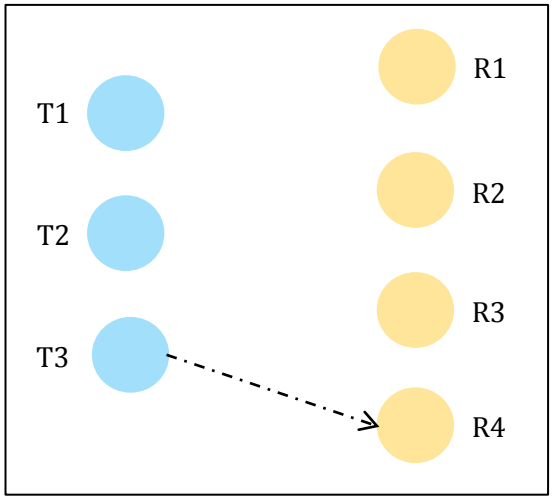
Şekil 3.12 H3 algoritması - İterasyon 1, T2'nin seçilmesi

- **İterasyon 1 (T1 için):** Kapsanan gereksinimler (R1) ve bu gereksinimlere izlenebilirlik veren diğer testler arasındaki bağlantılar silinir. R1 ile T2 hariç herhangi bir bağlantı olmadığı için sonraki adıma geçilir.
- **İterasyon 1 (T1 için):** Kalan gereksinimler üzerinden (R4), testlerin kapsama sayıları büyükten küçüğe sıralanır.

Çizelge 3.10 H3 İterasyon 1, kalan son testin gereksinim kapsama sayısı

Test	Kapsama Sayısı
T3	1

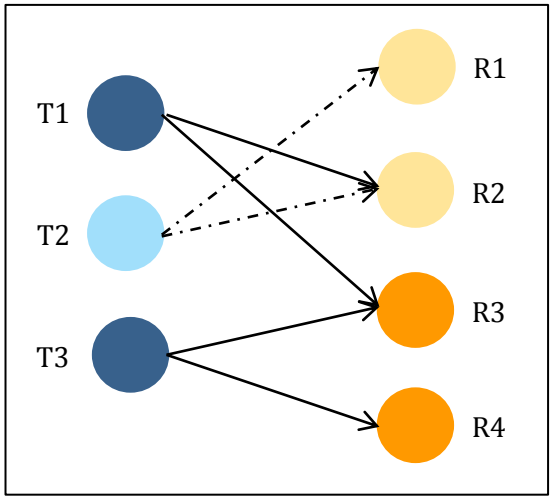
- **İterasyon 1 (T1 için):** İlk eleman olan T3 seçilir.
- **İterasyon 1 (T1 için):** T3 ve kapsadığı tüm gereksinimler arasındaki bağlantılar silinerek, bu gereksinimler, kapsanan gereksinimler arasına eklenir. T3, seçilen testlere eklenir.



Şekil 3.13 H3 algoritması - İterasyon 1, T3'ün seçilmesi

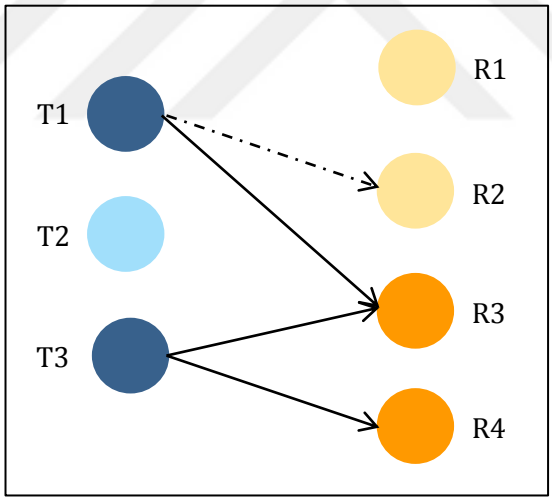
- **İterasyon 1 (T1 için):** Kapsanan gereksinimler (R4) ve bu gereksinimlere izlenebilirlik veren diğer testler arasındaki bağlantılar silinir. R4 ile T3 hariç herhangi bir bağlantı olmadığı için sonraki adıma geçilir.
- **İterasyon 1 (T1 için):** Tüm gereksinimler kapsandığı için iterasyon sonlanır.
- **İterasyon 1 (T1 için):** Seçilen Testler: T1, T2, T3, Testlerin sayısı:3
- **İterasyon 1 (T1 için):** $3 < \infty$ (GlobalCount) olduğu için, son durumda F içinde 3 test (T1,T2 ve T3) bulunmaktadır ve GlobalCount = 3 olarak güncellenir.

- **İterasyon 2 (T2 için):** T2 seçilir.
- **İterasyon 2 (T2 için):** T2 ve kapsadığı tüm gereksinimler arasındaki bağlantılar silinerek, bu gereksinimler, kapsanan gereksinimler arasına eklenir. T2, seçilen testlere eklenir.



Şekil 3.14 H3 algoritması - İterasyon 2, T2'nin seçilmesi

- **İterasyon 2 (T2 için):** Kapsanan gereksinimler (R1 ve R2) ve bu gereksinimlere izlenebilirlik veren diğer testler arasındaki bağlantılar silinir.



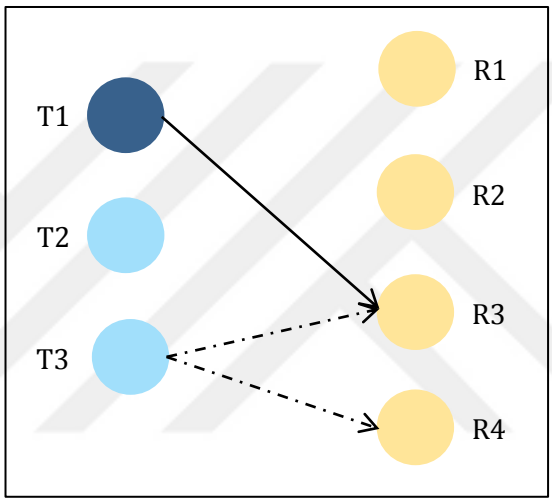
Şekil 3.15 H3 İterasyon 2, R1 ve R2'yi kapsayan testlerin bağlantılarının silinmesi

- **İterasyon 2 (T2 için):** Kalan gereksinimler üzerinden (R3 ve R4), testlerin kapsama sayıları büyükten küçüğe sıralanır.

Çizelge 3.11 H3 İterasyon 2, kalan testlerin gereksinim kapsama sayıları

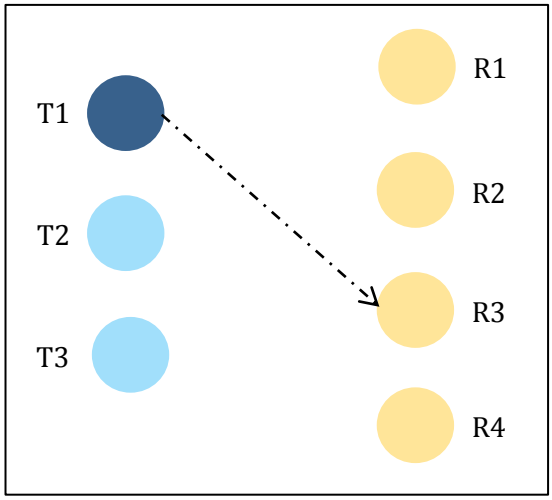
Test	Kapsama Sayısı
T3	2
T1	1

- **İterasyon 2 (T2 için):** İlk eleman olan T3 seçilir.
- **İterasyon 2 (T2 için):** T3 ve kapsadığı tüm gereksinimler arasındaki bağlantılar silinerek, bu gereksinimler, kapsanan gereksinimler arasına eklenir. T3, seçilen testlere eklenir.



Şekil 3.16 H3 algoritması - İterasyon 2, T3'ün seçilmesi

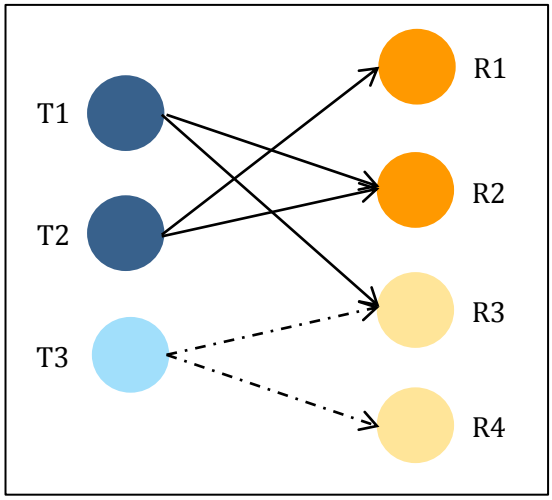
- **İterasyon 2 (T2 için):** Kapsanan gereksinimler (R3 ve R4) ve bu gereksinimlere izlenebilirlik veren diğer testler arasındaki bağlantılar silinir.



Şekil 3.17 H3 İterasyon 2, R3 ve R4'ü kapsayan testlerin bağlantılarının silinmesi

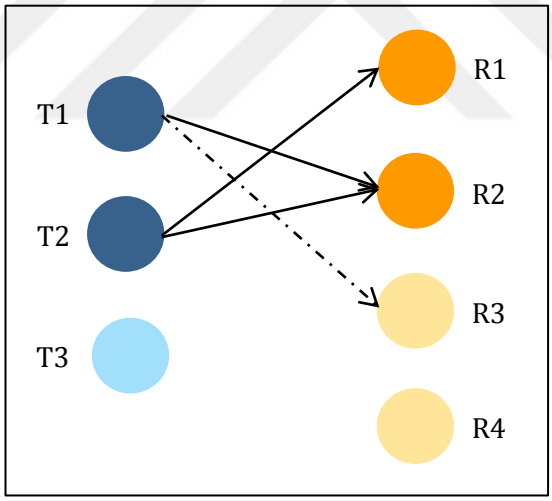
- **İterasyon 2 (T2 için):** Tüm gereksinimler kapsandığı için iterasyon sonlanır.
- **İterasyon 2 (T2 için):** Testler: T2, T3, Testlerin sayısı: 2
- **İterasyon 2 (T2 için):** 2 < 3 (GlobalCount) olduğu için, son durumda F içinde 2 test (T2 ve T3) bulunmaktadır ve GlobalCount = 2 olarak güncellenir.

- **İterasyon 3 (T3 için):** T3 seçilir.
- **İterasyon 3 (T3 için):** T3 ve kapsadığı tüm gereksinimler arasındaki bağlantılar silinerek, bu gereksinimler, kapsanan gereksinimler arasına eklenir. T3, seçilen testlere eklenir.



Şekil 3.18 H3 algoritması - İterasyon 3, T3'ün seçilmesi

- **İterasyon 3 (T3 için):** Kapsanan gereksinimler (R3 ve R4) ve bu gereksinimlere izlenebilirlik veren diğer testler arasındaki bağlantılar silinir.



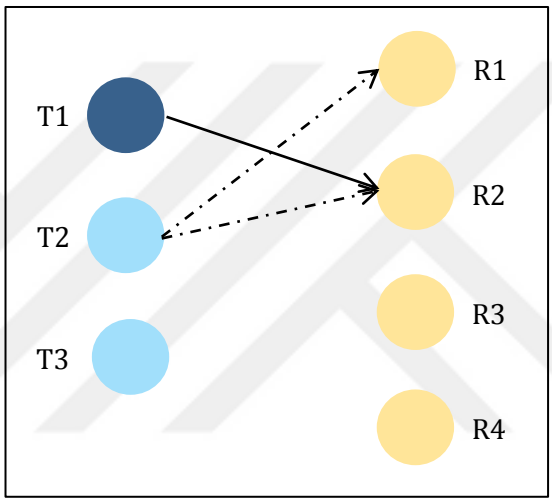
Şekil 3.19 H3 İterasyon 3, R3 ve R4'ü kapsayan testlerin bağlantılarının silinmesi

- **İterasyon 3 (T3 için):** Kalan gereksinimler üzerinden (R1 ve R2), testlerin kapsama sayıları büyükten küçüğe sıralanır.

Çizelge 3.12 H3 İterasyon 3, kalan testlerin gereksinim kapsama sayıları

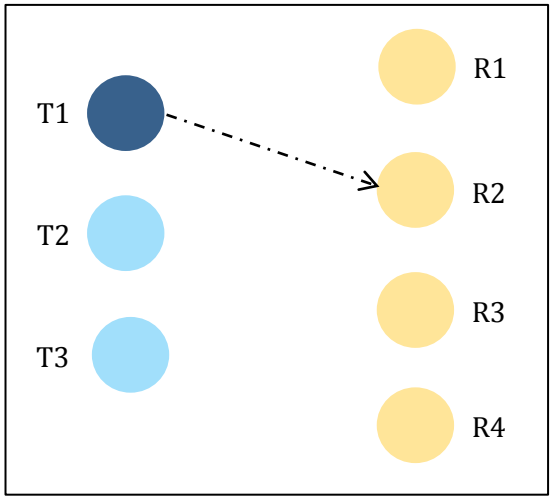
Test	Kapsama Sayısı
T2	2
T1	1

- **İterasyon 3 (T3 için):** İlk eleman olan T2 seçilir.
- **İterasyon 3 (T3 için):** T2 ve kapsadığı tüm gereksinimler arasındaki bağlantılar silinerek, bu gereksinimler, kapsanan gereksinimler arasına eklenir. T2, seçilen testlere eklenir.



Şekil 3.20 H3 algoritması - İterasyon 3, T2'nin seçilmesi

- **İterasyon 3 (T3 için):** Kapsanan gereksinimler (R1 ve R2) ve bu gereksinimlere izlenebilirlik veren diğer testler arasındaki bağlantılar silinir.



Şekil 3.21 H3 İterasyon 3, R1 ve R2'yi kapsayan testlerin bağlantılarının silinmesi

- **İterasyon 3 (T3 için):** Tüm gereksinimler kapsandığı için iterasyon sonlanır.
- **İterasyon 3 (T3 için):** Testler: T2, T3, Testlerin sayısı: 2
- **İterasyon 3 (T3 için):** 2 < 2 (GlobalCount) olmadığı için F üzerinde bir güncelleme yapılmaz.
- Son durumda F içinde 2 test (T2 ve T3) bulunmaktadır ve GlobalCount = 2 olarak güncellenmiştir.
- maxCoveringElements içindeki tüm elemanlar için işlem yapıldığı için program sonlanır.
- Seçilen Testler: T2, T3

Tüm algoritmalar verilen örnek üzerinde uygulandıktan sonraki durum Çizelge 3.13'de özetlenmiştir.

Çizelge 3.13 Algoritmaların son durumdaki karşılaştırmaları

Algoritmalar	Tüm Gereksinimleri Kapsamak İçin Gereken Test Sayısı	Testler
H-1	2	T2, T3
H-2	3	T1,T2,T3
H-3	2	T2,T3

4. GELİŞTİRİLEN ALGORİTMALARIN METASEZGİSEL ALGORİTMALARLA KARŞILAŞTIRILMASI

Geliştirilen H1, H2 ve H3 algoritmaları, Genetik Algoritma (GA) ve Karınca Koloni Algoritması (ACO) ile karşılaştırılmıştır. Aynı veri seti üzerinde 1, 15, 30 ve 60 dakikalık sürelerle algoritmalar çalıştırılmış ve sonuçlar tablolara yansıtılmıştır. Tablolarda belirtilen değerler saniye cinsindedir. İlk 6 veri seti rastgele üretilirken, sonraki 5 veri seti içinde çözümler barındıracak şekilde yine rastgele üretilmiştir. Bu sayede gereksinimleri kapsayabilecek minimum test durumu sayısı önceden tespit edilebilmiş ve GA ve ACO içinde kullanılan algoritmik değerler önceden verilebilmiştir.

Çizelge 4.1 Karşılaştırma tablosu - 1

# Toplam test durumu: 100 # Test durumundaki gereksinimler: 5 # Toplam gereksinim: 100 # Kapsanacak gereksinim: 15		**ALGORİTMA**				
		GA	**ACO**	**H1**	**H2**	**H3**
1	Son durumdaki test durumları	10	10	13	8	8
	Kapsanan gereksinimler	15	15	15	15	15
	Kapsanmayan gereksinimler	0	0	0	0	0
	Geçen Süre	1	35	0	0	5
15	Son durumdaki test durumları	10	10	13	8	8
	Kapsanan gereksinimler	15	15	15	15	15
	Kapsanmayan gereksinimler	0	0	0	0	0
	Geçen Süre	1	35	0	0	5
30	Son durumdaki test durumları	10	10	13	8	8
	Kapsanan gereksinimler	15	15	15	15	15
	Kapsanmayan gereksinimler	0	0	0	0	0
	Geçen Süre	1	35	0	0	5
60	Son durumdaki test durumları	10	10	13	8	8
	Kapsanan gereksinimler	15	15	15	15	15
	Kapsanmayan gereksinimler	0	0	0	0	0
	Geçen Süre	1	35	0	0	5

Çizelge 4.1’de belirtilen parametrelerle çalıştırıldığında performansı en yüksek algoritma, en az test durumu ile tüm gereksinimleri en az sürede kapsayabilen H2 algoritması olmuştur. En kötü algoritma ise gereksinimleri en uzun sürede kapsayabilen ACO olmuştur.

Çizelge 4.2 Karşılaştırma tablosu - 2

# Toplam test durumu: 1000 # Test durumundaki gereksinimler: 5 # Toplam gereksinim: 1000 # Kapsanacak gereksinim: 25		**ALGORİTMA**				
		GA	**ACO**	**H1**	**H2**	**H3**
1	Son durumdaki test durumları	20	20	25	20	20
	Kapsanan gereksinimler	25	5	25	25	25
	Kapsanmayan gereksinimler	0	20	0	0	0
	Geçen Süre	24	60	0	0	60
15	Son durumdaki test durumları	20	20	25	20	20
	Kapsanan gereksinimler	25	6	25	25	25
	Kapsanmayan gereksinimler	0	19	0	0	0
	Geçen Süre	347	900	0	0	900
30	Son durumdaki test durumları	20	20	25	20	20
	Kapsanan gereksinimler	25	14	25	25	25
	Kapsanmayan gereksinimler	0	11	0	0	0
	Geçen Süre	265	1800	0	0	1800
60	Son durumdaki test durumları	20	20	25	20	20
	Kapsanan gereksinimler	25	15	25	25	25
	Kapsanmayan gereksinimler	0	10	0	0	0
	Geçen Süre	943	3600	0	0	3600

Çizelge 4.2’de belirtilen parametrelerle çalıştırıldığında performansı en yüksek algoritma, en az test durumu ile tüm gereksinimleri en az sürede kapsayabilen H2 algoritması olmuştur. En kötü algoritma ise çözümü bulamayan ACO olmuştur.

Çizelge 4.3 Karşılaştırma tablosu - 3

# Toplam test durumu: 10000 # Test durumundaki gereksinimler: 25 # Toplam gereksinim: 10000 # Kapsanacak gereksinim: 100		ALGORİTMA				
		GA	**ACO**	**H1**	**H2**	**H3**
1	Son durumdaki test durumları	50	50	86	52	52
	Kapsanan gereksinimler	63	19	100	100	100
	Kapsanmayan gereksinimler	37	81	0	0	0
	Geçen Süre	60	60	19	24	60
15	Son durumdaki test durumları	50	50	86	52	52
	Kapsanan gereksinimler	93	19	100	100	100
	Kapsanmayan gereksinimler	7	81	0	0	0
	Geçen Süre	900	900	19	24	900
30	Son durumdaki test durumları	50	50	86	52	52
	Kapsanan gereksinimler	94	19	100	100	100
	Kapsanmayan gereksinimler	6	81	0	0	0
	Geçen Süre	1800	1800	19	24	1800
60	Son durumdaki test durumları	50	50	86	52	52
	Kapsanan gereksinimler	96	19	100	100	100
	Kapsanmayan gereksinimler	4	81	0	0	0
	Geçen Süre	3600	3600	19	24	3600

Çizelge 4.3'te belirtilen parametrelerle çalıştırıldığında performansı en yüksek algoritma, en az test durumu ile tüm gereksinimleri kapsayabilen H2 algoritması olmuştur. Tüm gereksinimleri en kısa sürede kapsayabilen H1 de performansı yüksek algoritmalardandır. En kötü algoritma ise çözüme en uzak gelişmeyi gösteren ACO olmuştur.

Çizelge 4.4 Karşılaştırma tablosu - 4

# Toplam test durumu: 1000 # Test durumundaki gereksinimler: 100 # Toplam gereksinim: 1000 # Kapsanacak gereksinim: 1000		ALGORİTMA				
		GA	**ACO**	**H1**	**H2**	**H3**
1	Son durumdaki test durumları	25	25	46	27	26
	Kapsanan gereksinimler	977	944	1000	1000	1000
	Kapsanmayan gereksinimler	23	56	0	0	0
	Geçen Süre	60	60	0	0	60
15	Son durumdaki test durumları	25	25	46	27	26
	Kapsanan gereksinimler	992	943	1000	1000	1000
	Kapsanmayan gereksinimler	8	57	0	0	0
	Geçen Süre	900	900	0	0	900
30	Son durumdaki test durumları	25	25	46	27	26
	Kapsanan gereksinimler	993	944	1000	1000	1000
	Kapsanmayan gereksinimler	7	56	0	0	0
	Geçen Süre	1800	1800	0	0	1800
60	Son durumdaki test durumları	25	25	46	27	26
	Kapsanan gereksinimler	992	945	1000	1000	1000
	Kapsanmayan gereksinimler	8	55	0	0	0
	Geçen Süre	3600	3600	0	0	3600

Çizelge 4.4'te belirtilen parametrelerle çalıştırıldığında performansı en yüksek algoritma, en az test durumu ile tüm gereksinimleri en az sürede kapsayabilen H2 algoritması olmuştur. En kötü algoritma ise çözüme en uzak gelişmeyi gösteren ACO olmuştur.

Çizelge 4.5 Karşılaştırma tablosu - 5

# Toplam test durumu: 10000 # Test durumundaki gereksinimler: 100 # Toplam gereksinim: 10000 # Kapsanacak gereksinim: 10000		ALGORİTMA				
		GA	**ACO**	**H1**	**H2**	**H3**
1	Son durumdaki test durumları	25	25	461	265	265
	Kapsanan gereksinimler	2292	2257	10000	10000	10000
	Kapsanmayan gereksinimler	7708	7743	0	0	0
	Geçen Süre	60	60	54	49	60
15	Son durumdaki test durumları	25	25	461	265	265
	Kapsanan gereksinimler	2392	2257	10000	10000	10000
	Kapsanmayan gereksinimler	7608	7743	0	0	0
	Geçen Süre	900	900	54	49	900
30	Son durumdaki test durumları	25	25	461	265	265
	Kapsanan gereksinimler	2406	2257	10000	10000	10000
	Kapsanmayan gereksinimler	7594	7743	0	0	0
	Geçen Süre	1800	1800	54	49	1800
60	Son durumdaki test durumları	25	25	461	265	265
	Kapsanan gereksinimler	2409	2257	10000	10000	10000
	Kapsanmayan gereksinimler	7591	7743	0	0	0
	Geçen Süre	3600	3600	54	49	3600

Çizelge 4.5'te belirtilen parametrelerle çalıştırıldığında performansı en yüksek algoritma, en az test durumu ile tüm gereksinimleri kapsayabilen H2 algoritması olmuştur. En kötü algoritma ise çözüme en uzak gelişmeyi gösteren ACO olmuştur.

Çizelge 4.6 Karşılaştırma tablosu - 6

# Toplam test durumu: 1000 # Test durumundaki gereksinimler: 10 # Toplam gereksinim: 1000 # Kapsanacak gereksinim: 1000		ALGORİTMA				
		GA	**ACO**	**H1**	**H2**	**H3**
1	Son durumdaki test durumları	50	50	235	156	155
	Kapsanan gereksinimler	457	413	1000	1000	1000
	Kapsanmayan gereksinimler	543	587	0	0	0
	Geçen Süre	60	60	1	1	60
15	Son durumdaki test durumları	50	50	235	156	155
	Kapsanan gereksinimler	491	412	1000	1000	1000
	Kapsanmayan gereksinimler	509	588	0	0	0
	Geçen Süre	900	900	1	1	900
30	Son durumdaki test durumları	50	20	235	156	155
	Kapsanan gereksinimler	495	411	1000	1000	1000
	Kapsanmayan gereksinimler	505	589	0	0	0
	Geçen Süre	1800	1800	1	1	1800
60	Son durumdaki test durumları	50	20	235	156	155
	Kapsanan gereksinimler	496	414	1000	1000	1000
	Kapsanmayan gereksinimler	504	586	0	0	0
	Geçen Süre	3600	3600	1	1	3600

Çizelge 4.6'da belirtilen parametrelerle çalıştırıldığında performansı en yüksek algoritma tüm gereksinimleri en az sürede kapsayabilen H2 algoritması olmuştur. Ayıca H3 algoritması da en az sayıda test durumu ile tüm gereksinimleri kapsadığı için performansı yüksek algoritmalar arasındadır. En kötü algoritma ise çözüme en uzak gelişmeyi gösteren ACO olmuştur.

Çizelge 4.7 Karşılaştırma tablosu - 7

# Toplam test durumu: 50 # Test durumundaki gereksinimler: 50 # Toplam gereksinim: 1000 # Kapsanacak gereksinim: 1000		ALGORİTMA				
		GA	**ACO**	**H1**	**H2**	**H3**
1	Son durumdaki test durumları	20	20	20	26	22
	Kapsanan gereksinimler	1000	744	1000	1000	1000
	Kapsanmayan gereksinimler	0	256	0	0	0
	Geçen Süre	19	60	0	0	60
15	Son durumdaki test durumları	20	20	20	26	22
	Kapsanan gereksinimler	1000	784	1000	1000	1000
	Kapsanmayan gereksinimler	0	216	0	0	0
	Geçen Süre	26	900	0	0	900
30	Son durumdaki test durumları	20	20	20	26	22
	Kapsanan gereksinimler	1000	797	1000	1000	1000
	Kapsanmayan gereksinimler	0	203	0	0	0
	Geçen Süre	18	1800	0	0	1800
60	Son durumdaki test durumları	20	20	20	26	21
	Kapsanan gereksinimler	1000	796	1000	1000	1000
	Kapsanmayan gereksinimler	0	204	0	0	0
	Geçen Süre	18	3600	0	0	3600

Çizelge 4.7'de belirtilen parametrelerle çalıştırıldığında performansı en yüksek algoritma, en az test durumu ile tüm gereksinimleri kapsayabilen H1 algoritması olmuştur. Çözümü kısa sürede bulabilen GA da performansı yüksek algoritmalardan biridir. En kötü algoritma ise çözüme en uzak gelişmeyi gösteren ACO olmuştur.

Çizelge 4.8 Karşılaştırma tablosu - 8

# Toplam test durumu: 100 # Test durumundaki gereksinimler: 10 # Toplam gereksinim: 100 # Kapsanacak gereksinim: 100		ALGORİTMA				
		GA	**ACO**	**H1**	**H2**	**H3**
1	Son durumdaki test durumları	10	10	22	17	15
	Kapsanan gereksinimler	92	75	100	100	100
	Kapsanmayan gereksinimler	8	25	0	0	0
	Geçen Süre	45	60	0	0	60
15	Son durumdaki test durumları	10	10	22	17	15
	Kapsanan gereksinimler	91	76	100	100	100
	Kapsanmayan gereksinimler	9	24	0	0	0
	Geçen Süre	542	900	0	0	900
30	Son durumdaki test durumları	10	10	22	17	12
	Kapsanan gereksinimler	97	77	100	100	100
	Kapsanmayan gereksinimler	3	23	0	0	0
	Geçen Süre	464	1800	0	0	1800
60	Son durumdaki test durumları	10	10	22	17	11
	Kapsanan gereksinimler	92	78	100	100	100
	Kapsanmayan gereksinimler	8	22	0	0	0
	Geçen Süre	2004	3600	0	0	3600

Çizelge 4.8'de belirtilen parametrelerle algoritmalar kısa süre çalıştırıldığında özellikle H1, H2 ve H3 yakın performanslar sergilerken, süre arttıkça H3 test durumu sayısını düşürebilmiştir. En kötü algoritma ise çözüme en uzak gelişmeyi gösteren ACO olmuştur.

Çizelge 4.9 Karşılaştırma tablosu - 9

# Toplam test durumu: 1000 # Test durumundaki gereksinimler: 40 # Toplam gereksinim: 1000 # Kapsanacak gereksinim: 1000		ALGORİTMA				
		GA	**ACO**	**H1**	**H2**	**H3**
1	Son durumdaki test durumları	25	25	92	58	56
	Kapsanan gereksinimler	736	661	1000	1000	1000
	Kapsanmayan gereksinimler	264	339	0	0	0
	Geçen Süre	60	60	0	0	60
15	Son durumdaki test durumları	25	25	92	58	56
	Kapsanan gereksinimler	784	662	1000	1000	1000
	Kapsanmayan gereksinimler	216	338	0	0	0
	Geçen Süre	834	900	0	0	900
30	Son durumdaki test durumları	25	25	92	58	56
	Kapsanan gereksinimler	759	663	1000	1000	1000
	Kapsanmayan gereksinimler	241	337	0	0	0
	Geçen Süre	1800	1800	0	0	1800
60	Son durumdaki test durumları	25	25	92	58	56
	Kapsanan gereksinimler	779	664	1000	1000	1000
	Kapsanmayan gereksinimler	221	336	0	0	0
	Geçen Süre	3285	3600	0	0	3600

Çizelge 4.9'da belirtilen parametrelerle algoritmalar kısa süre çalıştırıldığında H2 ve H3 yakın performanslar sergilerken, süre arttıkça H3 test durumu sayısını düşürebilmiştir. H2 ise kısa sürede çözüme ulaşabilmiştir. En kötü algoritma ise çözüme en uzak gelişmeyi gösteren ACO olmuştur.

Çizelge 4.10 Karşılaştırma tablosu - 10

# Toplam test durumu: 10.000 # Test durumundaki gereksinimler: 50 # Toplam gereksinim: 10.000 # Kapsanacak gereksinim: 10.000		ALGORİTMA				
		GA	**ACO**	**H1**	**H2**	**H3**
1	Son durumdaki test durumları	200	200	523	422	421
	Kapsanan gereksinimler	6435	6403	9283	9854	9856
	Kapsanmayan gereksinimler	3565	3597	717	146	144
	Geçen Süre	60	60	60	57	60
15	Son durumdaki test durumları	200	200	768	462	461
	Kapsanan gereksinimler	6625	6403	10000	10000	10000
	Kapsanmayan gereksinimler	3375	3597	0	0	0
	Geçen Süre	900	900	75	59	900
30	Son durumdaki test durumları	200	200	768	462	460
	Kapsanan gereksinimler	6655	6403	10000	10000	10000
	Kapsanmayan gereksinimler	3345	3597	0	0	0
	Geçen Süre	1800	1800	75	51	1800
60	Son durumdaki test durumları	200	200	768	462	460
	Kapsanan gereksinimler	6690	6404	10000	10000	10000
	Kapsanmayan gereksinimler	3310	3596	0	0	0
	Geçen Süre	3600	3600	75	63	3600

Çizelge 4.10'da belirtilen parametrelerle çalıştığında, H2 kısa sürede çözüme ulaşma, H3 ise en az test durumu kullanarak gereksinimleri kapsama ile performansları yüksek algoritmalardır. En kötü algoritma ise çözüme en uzak gelişmeyi gösteren ACO olmuştur.

Çizelge 4.11 Karşılaştırma tablosu - 11

# Toplam test durumu: 100.000 # Test durumundaki gereksinimler: 1000 # Toplam gereksinim: 100.000 # Kapsanacak gereksinim: 100.000		ALGORİTMA				
		GA	**ACO**	**H1**	**H2**	**H3**
1	Son durumdaki test durumları	100	100	0	0	0
	Kapsanan gereksinimler	63421	63616	0	0	0
	Kapsanmayan gereksinimler	36579	36384	100000	100000	100000
	Geçen Süre	60	60	60	60	60
15	Son durumdaki test durumları	100	100	0	0	0
	Kapsanan gereksinimler	63809	63616	0	0	0
	Kapsanmayan gereksinimler	36191	36384	100000	100000	100000
	Geçen Süre	900	900	900	900	900
30	Son durumdaki test durumları	100	100	31	32	32
	Kapsanan gereksinimler	63990	63616	26464	28888	28888
	Kapsanmayan gereksinimler	36010	36384	73536	71112	71112
	Geçen Süre	1800	1800	1800	1800	1800
60	Son durumdaki test durumları	100	100	523	414	414
	Kapsanan gereksinimler	64165	63616	97141	100000	100000
	Kapsanmayan gereksinimler	35835	36384	2859	0	0
	Geçen Süre	3600	3600	3600	3436	3436

Aşağıda, Çizelge 4.11'de gösterilen verilerin incelemesi yapılmıştır.

1. 1 dakikalık süre kısıtlaması ile çalıştırıldığında;
   - Hiçbir algoritma belirtilen süre içinde çözümü bulamamıştır.
   - H1, H2 ve H3 herhangi bir sonuç üretememiştir.
   - GA, 100 test durumu ile 63421 gereksinimi kapsayabilmiştir.
   - ACO, 100 test durumu ile 63616 gereksinimi kapsayabilmiştir.
2. 15 dakikalık süre kısıtlaması ile çalıştırıldığında;
   - H1, H2 ve H3 herhangi bir sonuç üretememiştir.

- GA belirtilen süre içinde çözümü bulamamıştır. Ancak kapsadığı gereksinim sayısını 63421'den 63809'a yükseltmiştir. Ortalama süre ise 60'tan 900'e yükselmiştir.
- ACO herhangi bir gelişme göstermemiştir. Ayrıca ortalama süre ise 60'tan 900'e yükselmiştir.

3. 30 dakikalık süre kısıtlaması ile çalıştırıldığında;
   - H1, 31 test durumu ile 26464 gereksinimi kapsamıştır.
   - H2, 32 test durumu ile 28888 gereksinimi kapsamıştır.
   - H3, 32 test durumu ile 28888 gereksinimi kapsamıştır.
   - GA belirtilen süre içinde çözümü bulamamıştır. Ancak kapsadığı gereksinim sayısını 63809'dan 63990'a yükseltmiştir. Ortalama süre ise 900'den 1800'e yükselmiştir.
   - ACO herhangi bir gelişme göstermemiştir. Ayrıca ortalama süre ise 900'den 1800'e yükselmiştir.
4. 60 dakikalık süre kısıtlaması ile çalıştırıldığında;
   - H2 ve H3 algoritmalarının 3436 saniyede çözümü bulduğu gözlemlenmiştir.
   - H2 ve H3, 414 test durumu ile sonuca ulaşmıştır.
   - H1, 523 test durumu ile 97141 gereksinimi kapsamıştır.
   - GA belirtilen süre içinde çözümü bulamamıştır. Ancak kapsadığı gereksinim sayısını 63990'dan 64165'e yükseltmiştir. Ortalama süre ise 1800'den 3600'e yükselmiştir.
   - ACO herhangi bir gelişme göstermemiştir. Ayrıca ortalama süre ise 1800'den 3600'e yükselmiştir.

Çizelge 4.11'de belirtilen parametrelerle algoritmalar kısa süre çalıştırıldığında ACO ve GA'nın performansı yüksek iken süre arttıkça H2 ve H3 çözüme ulaşmış, H1 ise çözüme yaklaşmıştır.

Tüm tablolar incelendiğinde aşağıdaki durumlar tespit edilmiştir:

- GA ve ACO'da sırasıyla kromozom ve yol uzunluğu değerleri önceden verilemediği için bazı durumlarda sonuç bulunamamıştır.
- GA ve ACO bazı durumlarda süre artmasına rağmen daha kötü sonuç üretmiştir.
- Süre artışı ile sonucu iyileştirme garantisi olan tek algoritma H3'tür.
- H3 süre artışına rağmen çoğunlukla H2 ile aynı sonuçları üretmiştir.
- H3, H2 ile aynı anda sonuca ulaşabilmiştir. Ancak iterasyonlara devam ettiği için ortalama süresi artış göstermiştir.
- H3'te çoğunlukla, zaman artmasına rağmen gelişme olmamış. sadece süre artışı meydana gelmiştir.
- Büyük parametrelerde iterasyonlar uzun zaman aldığı için ortalama süreler de artmıştır. Ancak bu durumdan en çok etkilenen algoritma ACO olmuştur.
- Büyük parametrelerde, kısa sürede H1, H2 ve H3 sonuç üretememeye başlamıştır.
- H1, H2 ve H3 deterministik algoritmalar olduğu için her çalışmada aynı sonucu üretmiştir.
- Veri seti içine çözüm eklenmediği durumlarda H2 algoritması ön plana çıkarken, çözüm eklendiğinde diğer algoritmalar da iyi sonuçlar üretebilmiştir.

5. SONUÇ

Yapılan bu araştırmada yazılım yaşam döngüsü hakkında bilgi verilmiş ve bu döngünün önemli bir adımı olan test aşaması incelenmiştir. Yazılımı test ederken kullanılan farklı test metodolojileri ile ilgili örnekler verilmiştir. Bu metodolojilerden biri olan regresyon testlerinin kullanım amaçları, bu testler yapılırken kontrol edilen durumlar, regresyon testi yöntemleri ve test sırasında oluşabilecek zorluklar belirtilmiştir. Regresyon testinin daha etkin yapılmasını sağlayan yöntemlerden biri olan test durumu önceliklendirmenin kullanım amacı açıklanmıştır. Test durum önceliklendirme kapsamında kullanılan algoritmalar örnekler eşliğinde incelenmiş ve hakim küme yöntemi kullanılarak 3 yeni algoritma geliştirilmiştir. Bu geliştirilen algoritmalar, metasezgisel algoritmalar ile karşılaştırılmış, aynı veri seti kullanılarak performans analiz yapılmıştır.

Yeni geliştirilen algoritmalar, metasezgisel algoritmalar ile karşılaştırılmış ve aynı veri setleri kullanılarak performans analizleri yapılmıştır. Yapılan karşılaştırmalar sonucunda, içerisinde çözüm barındırmayan ilk 6 veri setinde H2 algoritması en iyi performansı sergilemiştir. Bu veri setleri, içinde çözüm içermediği için GA ve ACO için gerekli parametreler verilememiştir. Bu durum algoritmaların performansını negatif yönde etkilemiştir. İçerisinde çözümleri içeren son 5 veri setinde ise yine H2 algoritması ön plana çıkarken, metasezgisel algoritmaların performansı da artış göstermiştir. Geliştirilen algoritmaların tümü problem çözümünü her veri setinde sağlamıştır. Ancak veri seti büyüdükçe kısa sürelerde çözüm üretememeye başlamışlardır. Büyük veri setlerinde metasezgisel algoritmaların kısa süre performansının daha iyi olduğu gözlemlenmiştir. Veri seti büyüdükçe ACO'nun performansında diğer algoritmalara göre daha büyük bir düşüş yaşanmıştır. H1 algoritması belli veri setlerinde en iyi performansı gösterirken çoğu veri setinde H2'nin gerisinde kalmıştır. H3 algoritması çoğunlukla H2 ile aynı sonucu üretse de ortalama süresinin daha fazla olduğu belirlenmiştir. Bunun yanında sonucu iyileştirme garantisi olan tek algoritmanın H3 olduğu tespit edilmiştir.

KAYNAKLAR

Akdeniz, E. 2017. Web Sitesi: https://btisanalisti.com/yazilim-test-muhendisligi-onemini-artiriyor/, Erişim Tarihi: 06.03.2021.

Anonymous. 2019. Web Sitesi: https://en.wikipedia.org/wiki/List_of_software_bugs, Erişim Tarihi: 06.03.2021.

Ansari, A., Khanb, A., Khanc, A. and Mukadamd, K. 2016. Optimized Regression Test using Test Case Prioritization. Procedia Computer Science, 79, 152–160.

Azizi, M. And Do, H. 2018. Graphite: A Greedy Graph-Based Technique for Regression Test Case Prioritization, IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 15-18 October, 245-251, Memphis, TN, USA.

Beyhan, F. 2018. Web Sitesi: https://medium.com/@frknbyhn/yaz%C4%B1l%C4%B1m-testi-genel-konular-4e408105529f, Erişim Tarihi: 06.03.2021.

Chi, J., Qu Y., Zheng Q., Yang Z., Jin W., Cui D. and Liu T. 2020. Relation-based Test Case Prioritization For Regression Testing, Journal of Systems and Software, 163.

Çetin, Ö.H. 2019. Web Sitesi: https://medium.com/@omerharuncetin/yaz%C4%B1l%C4%B1m-ya%C5%9Fam-d%C3%B6ng%C3%BC-modelleri-543c7879a742 , Erişim Tarihi: 06.03.2021.

Farooq, F. and Nadeem, A. 2017. A Fault Based Approach to Test Case Prioritization, International Conference on Frontiers of Information Technology (FIT), 18-20 December, 52-57, Islamabad, Pakistan.

Gokilavani, N. and Bharathi, B. 2021. Test Case Prioritization To Examine Software For Fault Detection Using PCA Extraction And K-means Clustering With Ranking. Soft Computing, 25, 5163–5172.

Hla, K. H. S., Choi, Y. and Park, J. S. 2008. Applying Particle Swarm Optimization to Prioritizing Test Cases for Embedded Real Time Software Retesting, IEEE 8th International Conference on Computer and Information Technology Workshops, 8-11 July, 527-532, Sydney, Australia.

Jeffrey, D. and Gupta, N. 2006. Test Case Prioritization Using Relevant Slices, 30th Annual International Computer Software and Applications Conference (COMPSAC'06), 17-21 September, 411–420, Chicago, USA.

Jun, W., Yan, Z. and Chen, J. 2011. Test Case Prioritization Technique based on Genetic Algorithm, International Conference on Internet Computing and Information Services, 17-18 September, 173–175, Hong Kong, China.

Kaur, A. and Bhatt, D. 2011. Hybrid Particle Swarm Optimization for Regression Testing, International Journal on Computer Science and Engineering, 3(5), 1815–1824.

Kavitha, R., Kavitha, V. R. and Suresh Kumar, N. 2010. Requirement Based Test Case Prioritization, International Conference On Communication Control And Computing Technologies, 7-9 October, 826-829, Ramanathapuram, India.

Kayes, M. I. 2011. Test Case Prioritization For Regression Testing Based on Fault Dependency, 3rd International Conference on Electronics Computer Technology, 8-10 April, 48-52, Kanyakumari, India.

Kennedy, J., and Eberhart, R. 1995. Particle Swarm Optimization, Proceedings of ICNN'95 - International Conference on Neural Networks, 27 November-1 December, 1942-1948, Perth, WA, Australia.

Kılınç, D. 2016. Web Sitesi: https://medium.com/@denizkilinc/yaz%C4%B1l%C4%B1m-ya%C5%9Fam-d%C3%B6ng%C3%BCs%C3%BC-temel-a%C5%9Famalar%C4%B1-software-development-life-cycle-core-processes-197a4b503696 , Erişim Tarihi: 06.03.2021.

Kim, J. and Porter, A. 2002. A History-Based Test Prioritization Technique for Regression Testing in Resource Constrained Environments, Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), 25-25 May, 119-129, Orlando, USA.

Konsaard, P. and Ramingwong, L. 2015. Total Coverage Based Regression Test Case Prioritization using Genetic Algorithm, 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), 24-27 June, 1–6, Hua Hin, Thailand.

Krishnamoorthi, R., Sahaaya, S.A. and Mary, S.A. 2009. Regression Test Suite Prioritization using Genetic Algorithms, International Journal of Hybrid Information Technology, 2(3), 35–52.

Li, Z., Harman, M. and Hierons, R.M. 2007. Search Algorithms for Regression Test Case Prioritization, IEEE Transactions On Software Engineering, 33(4), 225–237.

Liedlof, M. 2008. Finding a dominating set on bipartite graphs, Information Processing Letters, 107(5), 154-157.

Lin, C., Chen, C., Tsai, C. and Kapfhammer, G. M. 2013. History-Based Test Case Prioritization with Software Version Awareness, 18th International Conference on Engineering of Complex Computer Systems, 17-19 July, 171-172, Singapore, Singapore.

Luo, L. 2001. Software Testing Techniques, Institute For Software Research International Carnegie Mellon University Pittsburgh, USA.

Malik, K.I., Khalid, H., Hassan, S. and Fatima, K. 2013. Software Testing Methodologies for Finding Errors, Research Journal of Social Science & Management, 3(7), 7-14.

Mohapatra, S. K. and Prasad, S. 2013. Evolutionary Search Algorithms for Test Case Prioritization, International Conference on Machine Intelligence and Research Advancement, 21-23 December, 115-119, Katra, India.

Murali, R., Koyutürk, M., Ananth, G. and Suresh, J. 2008. PHALANX: A Graph-theoretic Framework For Test Case Prioritization, Proceedings of the ACM Symposium on Applied Computing, 16-20 March, 667-673, Fortaleza, Ceara, Brazil.

Navdeep, Jain, C. and Gambhir, A. 2015. Cost Effective Regression Testing, Cognition, 4, 8-10.

Rothermel, G., Untch R. H., Chu, C. and Harrold, M. J. 1999. Test Case Prioritization: An Empirical Study. Proceedings IEEE International Conference on Software Maintenance, 30 August-3 September, 179-188, Oxford, England.

Singh, Y., Kaur, A and Suri, B. 2010. Test Case Prioritization using Ant Colony Optimization, Software Engineering Notes, 35(4), 1–7.

Srivastava, P.R. 2008. Test Case Prioritization, Journal of Theoretical and Applied Information Technology, 4, 178–181.

Wang, Y., Zhao, X. and Ding, X. 2015. An Effective Test Case Prioritization Method Based on Fault Severity. 6th IEEE International Conference on Software Engineering and Service Science (ICSESS), 23-25 September, 737-741, Beijing, China.