ANKARA ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

YÜKSEK LİSANS TEZİ

GENETİK PROGRAMLAMA İLE BİRİM-TESTLERİNDEN GO KODUNUN İLK TASLAĞININ ÜRETİMİ

Ufuktan YILDIRIM

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

ANKARA
2024

ÖZET

Yüksek Lisans Tezi

GENETİK PROGRAMLAMA İLE BİRİM-TESTLERİNDEN
GO KODUNUN İLK TASLAĞININ ÜRETİMİ

Ufuktan YILDIRIM

Ankara Üniversitesi
Fen Bilimleri Enstitüsü
Bilgisayar Mühendisli Anabilim Dalı

Danışman: Prof. Dr. Semra GÜNDÜÇ

Genetik programlama bilgisayar programlarının rastgele uygulanan değişimler ve uyuma dayalı elemelerin tekrarıyla yani evrimsel süreçle elde edildiği bir program sentezi yöntemidir. Ancak genetik programlama 30 yılı aşkın geçmişine rağmen henüz pek çok modern programlama diline uyarlanamamıştır. Sebebi ise bu dillerin zengin özellik seti ve çok aşamalı test süreçlerinin çözüm uzayını en iyi evrimsel aramanın dahi yolunu kaybetmeden, yerel minimumlara takılmadan çözümü bulamayacağı kadar genişletmesidir. Bu çalışmanın amacı LLM'lerin kod üretim özelliğinin gördüğü geniş ilginin sebep olacağı öngörülen zorunlu ve derlemeli dillerde kod sentezi yöntemleri araştırmalarındaki hızlanmaya katkı sağlaması için literatürde ve deneylerde karşılaşılmış sorunların ve onlara geliştirilen çözüm önerilerinin paylaşılmasıdır. Çalışmada tanıtılan yöntemlerden biri olan Katmanlı Arama, ana aramanın keşifsel karakterini korumak hedefiyle çalışma zamanı, sözdizimi ve basım hatalarına sahip denekler ortaya çıktığı her seferde, çok adımlı tamirleri bulmak için sınırlandırılmış, sığ, sıkışık ve izole alt evrimler başlatır. Ana aramayı sadece semantik hataların giderimine göre yapılandırmaya imkan tanır. Bu sayede, davranışı iyileştirmek için birden fazla yapısal düzenleme gerektiren yeniliklerin prematüre elenmekten korunması hedeflenir. Paylaşılan bir başka yöntem ST-ASTGP, AST kaynaklı çözüm uzayı genişlemesini azaltma hedefiyle sadece genotipte değil aynı zamanda fenotipte de tip güvenliğini gözeterek daha az söz dizimi hatasına sahip daha iyi denekler üretir.

Ocak 2024, 84 sayfa

Anahtar Kelimeler: Genetik Programlama, Kod Sentezi, Birim testi

ABSTRACT

Master Thesis

PRODUCING FIRST DRAFT OF GO CODE FROM UNIT-TESTS
WITH GENETIC PROGRAMMING

Ufuktan YILDIRIM

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

Supervisor: Prof. Dr. Semra GÜNDÜÇ

Genetic Programming is a program synthesis method that aims to find computer programs by repeatedly introducing randomly developed candidates and eliminating worser ones for generations. Yet, even after its 30 years long journey, there have been no success on applying it to an imperative and compiled language. Because of those languages' rich feature set and multi step testing process leading the solution space to expand further than the best evolutionary search can reach result before losing its way, stuck in local minima. Purpose of this work is to contribute into acceleration in code synthesis research on imperative and compiled languages which is expected to happen in public's recent interest on code producing abilities of LLMs by sharing solution ideas to current problems defined in literature and experimented in the research. One method introduced, Layered Search aims to protect explorative characteristic of the main search by spinning off overly restricted, shallower, dense and isolated sub-evolutions to quickly check possible, near fixes whenever a candidate with runtime, syntax or print errors is appeared in the main search. It enables main search to be tuned only for fixing semantic errors. This way, any innovation which needs multiple generations of structural improvement to improve behavior can get a chance against premature elimination. Another method ST-ASTGP targets reduction in AST-caused expansion in solution space by producing better candidates with less syntax errors by guarding type safety not only in the genotype but also in the phenotype as defined in language specification.

January 2024, 84 pages

Key Words: Genetic Programming, Code Synthesis, Unit-Testing

TEŞEKKÜR

İlk olarak danışmanım Sayın Prof. Dr. Semra GÜNDÜÇ'e (Ankara Üniversitesi Bilgisayar Mühendisliği Anabilim Dalı) tez sürecince sağladığı rehberlik, öneriler ve tüm destek için sonsuz teşekkürlerimi sunarım.

Tezin oluşumunda yardımcı olan aileme teşekkür ederim. Sabırları, destekleri ve cesaretlendirmeleri olmadan bu yolculuk çok daha zorlu olurdu.

Son olarak, bu tezin oluşturulmasında emeği geçen herkese teşekkür ederim. Umarım bu çalışma alanda yapılacak bilimsel çalışmalara bir katkı sağlar ve yeni ufuklar açar.

Ufuktan YILDIRIM
Ankara, Ocak 2024

İÇİNDEKİLER

KISALTMALAR DİZİNİ

ADF	Automatically Defined Functions
ANN	Artificial Neural Network
AST	Abstract Syntax Tree
ASTGP	Abstract Syntax Tree Genetic Programming
CA-STG	Context-Aware Subtree Generation
CEC	Congress on Evolutionary Computation
CI/CD	Continuous Integration / Continuous Deployment
CMA-ES	Covariance Matrix Adaptation Evolution Strategy
CRUD	Create Read Update Delete
EA	Evolutionary Algorithms
GA	Genetic Algorithm
GECCO	Genetic and Evolutionary Computation Conference
GEP	Gene Expression Programming
GP	Genetic Programming
HOTGP	Higher-Order Typed Genetic Programming
LLM	Large Language Model
NE	Neuroevolution
NEAT	Neuroevolution of Augmented Topologies
PSB2	Program Synthesis Benchmark Suite 2
REST API	Representational State Transfer Application Programming Interface
STG	Subtree Generation
STGP	Strongly Typed Genetic Programming
ST-ASTGP	Strongly Typed ASTGP
TDD	Test-Driven Development
TDE	Test-Driven Evolution

ŞEKİLLER DİZİNİ

ÇİZELGELER DİZİNİ

1. GİRİŞ

Literatüre bakıldığında görülmüştür ki, ortaya çıkışı 30 yılı aşmış olan ve bu süreçte pek çok farklı dilde bilgisayar programının kullanıcıdan alınan örnek girdi çıktı değerleri ile üretilmesine olanak tanımış Genetik Programlama (GP) henüz zorunlu ve derlemeli yapıdaki programlama dillerine uyarlanabilmiş değildir. Hedefe en çok yaklaşmış çalışmalar, zorunlu ve derlemeli bir dilde bir geliştirici tarafından yazılmış ancak hatalı çalışan kod üzerinde asgari düzenlemelerle tamirler gerçekleştiren çalışmalar ve sıfırdan kod üretimini fonksiyonel dillerde sağlayan çalışmalardır.

Bu tez için araştırmaya başlarken, nihayetinde bir geliştirici tarafından sağlanacak birim-testlerine uygun kodu zorunlu ve derlemeli bir dilde üretmeyi mümkün kılacak bir GP uyarlamasının tasarlanması hedeflenmiştir. Bu hedefe ulaşan bir çalışmanın literatürde bulunmamasıyla birlikte, çalışma kapsamında da ulaşılabilmiş değildir. Ancak ilerlemenin önündeki engeller saptanmış ve çözüm önerileri paylaşılmıştır.

Çalışmada Evrimsel Algoritma (EA) literatüründeki pek çok farklı alandan araştırmada paylaşılan yöntemler, hedef kitlenin ihtiyaçlarına ve imkanlarına uygun, benimseme maliyeti düşük bir araç geliştirmek için bir araya getirilmiştir.

1.1 Yöntem

Tanıtılan yöntem ifade ağaçları yerine Soyut Sözdizimi Ağacı yani AST (Abstract Syntax Tree) ve başarım fonksiyonları yerine birim-testleri ile çalışmayı hedeflemektedir. Rekabet eden denekleri temsil edebilmek için AST'ler kullanılmıştır (ASTGP). Bu yöntemde genetik operasyonlar (düzenlemeler) AST üzerinde gerçekleştirilmektedir. Deneklerin beklenen kriterleri sağlama oranının ölçümü için çok adımlı bir süreç yürütülür. AST'ler koda dönüştürülür, çalıştırıldığında birim testleri çalıştıracak testçi paket yerleştirilir, kod programa derlenir, testçi program çalıştırılır. Kaynak kodu ile AST arasındaki çift yönlü dönüşümleri gerçekleştirmek ve üretilen kod

tarafından erişilebilecek sembolleri listelemek için Go dili ile birlikte kullanılan statik analiz araçları kullanılmıştır.

Ana evrimsel aramanın koddaki semantik hataları gidermek için baskı, sözdizimi ve çalışma zamanı hatalı deneklerden arındırılmış çözüm uzayı üzerinde gezinebilmesini sağlamak için Katmanlı Arama algoritması tasarlanmış ve tanıtılmıştır. Bu algoritma sayesinde ana aramanın keşifsel, yerel minimumlara takılma ihtimali düşük karakterini korumak hedeflenmiştir. Ana arama N adımlı iyileştirmeleri (kompleks adaptasyonları) mümkün kılabilmek için ana arama sırasında ortaya çıkan hatalı denekler için sığ ve çok kollu, sınırlandırılmış, hızlı ve izole alt aramalar başlatır.

Tanıtılan başka bir yöntem ST-ASTGP, ifade ağaçları üzerindeki mutasyonlarda ast-üst tip uyumunu gözeterek çözüm uzayını daraltmayı sağlayan STGP'yi tip güvenliğini hem AST içindeki ast-üstler arasında hem de AST'nin temsil ettiği Go kodunun Go dilinin tip kurallarına uygunluğunu ASTGP'ye uyarlayarak AST kaynaklı çözüm uzayı genişlemesini baskılamayı hedefler.

1.2 Amaç

Bu çalışmanın amacı 92'den bu yana akademik çevrelerde gelişimini sürdürmüş GP'nin kod sentezi çözümlerinin hedef kitlesi çevrelere yayılmasını engellemiş eksikliklerini tanımlamak ve bunlara çözüm aramaktır. Son bir yılda LLM'lerin kod üretim yeteneklerinin geniş kitleler tarafından ilgiyle karşılanmasının ardından hızlanması beklenen kod sentezi araştırmalarına katkı sağlaması beklenmektedir.

1.3 Önem

ASTGP'nin yani denek temsili için Koza tarzı ifade ağaçları yerine AST'nin kullanımının neden çözüm uzayının aşırı büyümesine sebep olduğu saptanmış ve paylaşılmıştır.

AST kod dönüşümleri ve sembol filtrelemeleri için kullanılan statik analiz araçlarının pek çok dil için mevcut olması önerilen yöntemin farklı diller için uyarlanabilirliğini artırmaktadır.

Çalışmada önerilen yöntem zaten mevcutta var olan birim testlerini kullanmayı ve bu sayede kullanıcılar için benimseme maliyetini düşürmeyi hedefler. Yöntem hedeflenen haline ulaştığında, birim testleri, yöntemin ihtiyaç duyduğu tüm alana özel bilgiyi kullanıcıdan almakta yeterli olacağı için GP'nin iç işleyişi kullanıcıdan soyutlanacaktır dolayısıyla, çalışanların ayrıca eğitimine ihtiyaç duyulmayacaktır. Bununla birlikte birim testleri geliştirme bütçesi artırılabileceğinden dolayı TDD'nin faydalarından daha çok etkilenilecektir.

Özet olarak GP, çözümün formuna dair öngörüye sahip olunmadığında, özgün sorunlara çözüm üretmede, problemin maddi değeri hesaplama maliyetini aştığında tercih edilebilir bir yöntemdir. Arama uzayını daraltmak için hem alana özel hem de genel yöntemler bulunmaktadır. Artan işçi ücretleri ve düşen donanım fiyatları GP'nin uzun vadede lehine gözükmektedir.

2. KAVRAMLAR

2.1 Evrimsel Algoritmalar

Evrimsel algoritmalar geçmişte farklı alanlarda kullanılarak geçerliliklerini ispatlamışlardır. Örneğin NASA mühendisleri ST5 görevindeki uydular arasındaki iletişim için kullanılacak antenleri evrimsel algoritma yardımı ile elde etmiş ve tasarım ürünü olan QHA tipi antenle kıyaslayarak güç tüketimi, imalat karmaşıklığı, görev başarımı alanlarında daha iyi olduğunu görmüştür (Lohn vd. 2004, Hornby vd. 2006). GECCO (Genetic and Evolutionary Computation Conference), CEC (Congress on Evolutionary Computation), EvoStar gibi konferanslarda evrimsel algoritmaların çeşitli alanlarda gerçekleştirilen uygulamaları tanıtılır.

Geçmiş çalışmalar incelendiğinde EA alanındaki ilerlemelerin sonuna gelinmediği, ancak çeşitlenerek ve hızlanarak devam ettiği de görülebilir. EA olan pek çok yöntem benzerlikleriyle diğerlerinden farklılaşmıştır. Bunlardan bazıları yapay sinir ağlarını EA ile elde etmeyi hedefleyen Neuroevolution (NE), üretilecek denekleri daha kapsamlı bir stratejiyle belirleyen Evolution Strategies (ES), denekleri onları çaprazlamaya imkan sunan temsillerle tutan GA, EA'yı bilgisayar programlarını aramak için kullanan GP'dir.

Bu alanlardan biri olan Neuroevolution alanındaki çözümler incelendiğinde NEAT (Stanley ve Miikkulainen 2002) evrimsel algoritmanın getirdiği imkanları ortaya koymasıyla dikkat çekmektedir. Evrimsel arayışa ağın topolojisini de dahil eden NEAT hedef çözümü gereksiz özelleştirmelerden arındırılmış (generalization) verimli yapıyla elde eder, bu modellerin çalıştırılması da daha az hesaplama gerektirir.

2.1.1 Evrimsel Stratejiler

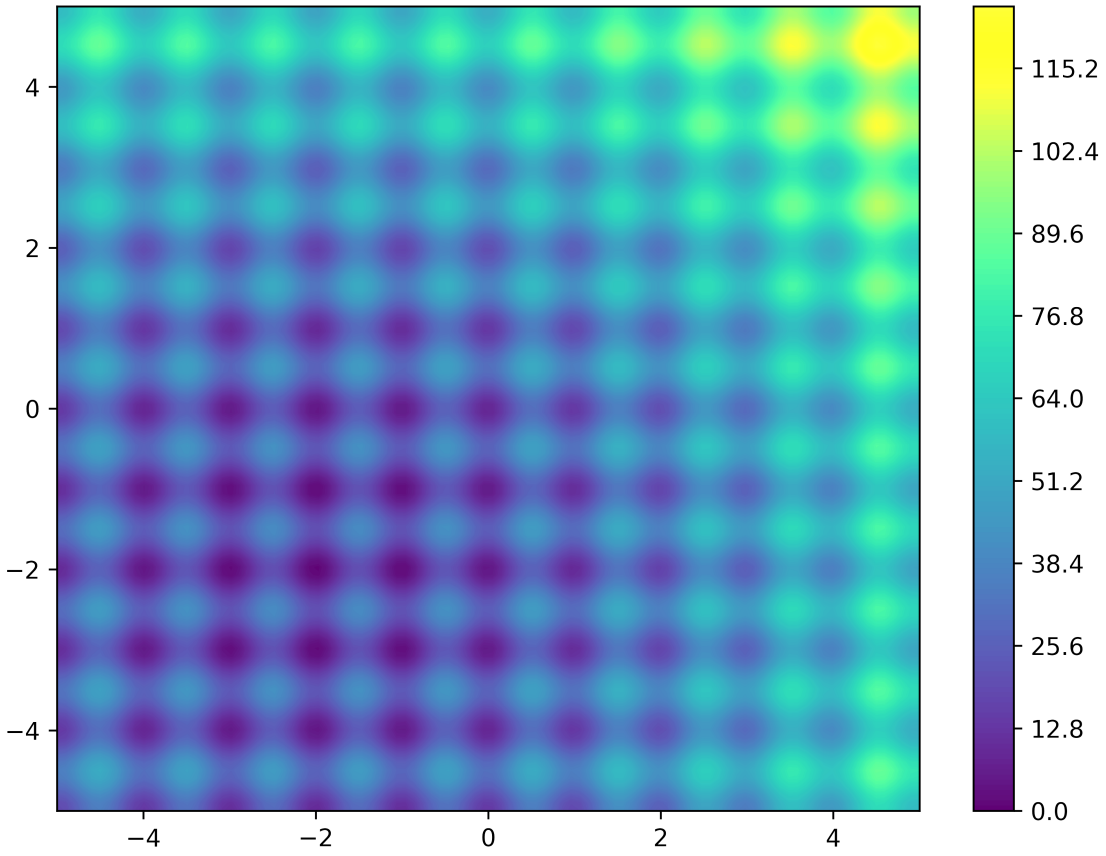
Bir EA'nın arama uzayı üzerindeki ilerleyişini tasarlamak mümkündür. Bu amaçta kullanılan yöntemlere Evolutionary Strategies (ES) denir. ES her yinelemede merkezi önceki nesildeki en başarılı olan bölgeye ilerleyerek sonuca daha hızlı ulaşmayı hedefler. CMA-ES her yinelemede arama uzayını daraltan veya genişleten bir ES algoritmasıdır. CMA-ES daraltma/genişletmeye karar vermek için kovaryans matrisinden yararlanır (Ha 2017, Anonymous 2021).

2.1.2 Genetik Algoritma

GA EA'nın özel bir türüdür. Bu türde denekler özel bir kodlamayla temsil edilir. Bu kodlama iki denek arasında EA'dan farklı olarak çaprazlama yapılmasına imkan tanır.

2.1.3 Yerel ve global minimumlar

Çözüm uzayının arama algoritması tarafından gezilen, ölçümlenen, ortaya çıkartılan alt kümesi arama uzayıdır. Çoğu sorun için ilgili çözüm uzayının pek çok yerel minima içermesi beklenir. Yerel minimumlar arama algoritmaları için yanıltıcıdır çünkü sürekli olarak daha uygun çözümleri arayan bir algoritma yerel minimumlara çekilir ve orada takılır. Bu yüzden global minimumlara erişemez. Örneğin Şekil 2.1'de pek çok yerel minimumu olan bir fonksiyon gösterilmiştir. Bu fonksiyonun global minimumu 0 değerini ürettiği (-2, -2) noktasıdır. Yakınındaki birkaç noktadan en düşük değerlisine gitmekten ibaret basit bir arama algoritması (-0.5, -0.5) noktasından aramaya başlarsa (-1, -1) noktasında geldiğinde bir sonraki düşük değerli noktaya (-2, -1) ile arasında bulunan (-1.5, -1) eşiğini aşamayacaktır.



Şekil 2.1 Pek çok yerel minimuma sahip bir çözüm uzayı

Genel olarak makine öğrenmesi yöntemleri veri setini parametrelerin daha doğru değerlerini oluşturmak için kullanır. Örneğin Gradient Descent modelin üretmesi gereken değerler ile ürettiği değerler arasındaki farkı kullanarak son katmandan başlayarak üretilen değerlerdeki hatalara daha fazla katkı sağlayan düğümlerin ağırlıklarını düşürür.

Evrimsel Algoritmaların yerel minimalara takılma ihtimali daha düşüktür. Çünkü veri setini ve hata bilgisini modeli düzenlemek için kullanmazlar; aramayı daha uygun sonuçların alındığı bölgede yoğunlaştırmak için kullanırlar.

EA'nın derin-öğrenme ve diğer veri seti tarafından yönlendirilen yapay zeka yöntemlerinden farklı olarak global maksimumu keşfetme ihtimalinin yüksekliğinden

gelen avantajı ve evrimsel algoritmaların en büyük dezavantajı olan yüksek işlem maliyetinin gelişen teknolojiyle birlikte azalacağı göz önünde bulundurulduğunda gelecek vadeden alternatiflerden biri olduğu düşünülmektedir.

2.1.4 Arama karakteristiği

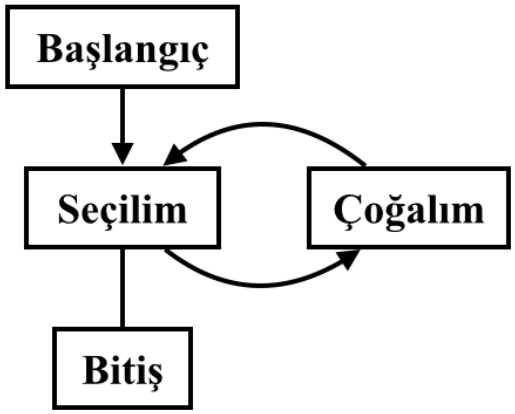
Dar bölgelerde yoğunlaşan arama algoritmaları sömürüsel karakterdedir. Bu aramalar mevcut dalları ilerletmekten daha çok yeni dallar oluşturur. Keşifsel aramalar çözüm uzayının geniş alanlarını kabaca taramayı daha kısa sürede gerçekleştirir. Yeni dallar oluşturmaktan çok mevcut dallarda derinleşilir.

Črepinšek vd. 2013 yılında yayınlanan araştırmalarında düzenli olarak uzak ve yeni alanlara ilerleyen davranışı keşifsel, komşu bölgelere ilerleyen davranışı sömürüsel tanımlamıştır. Sloss ve Gustafson'a (2019) göre çaprazlamalar mutasyonlara kıyasla çözüm uzayında daha büyük adımlarla ilerlemeyi sağlar. Dolayısıyla çaprazlamalar global aramalar için daha anlamlıyken mutasyonlar da yerel aramalar için daha uygundur. Yazarlara göre bir aramanın sömürüsel/keşifsel davranışı iyi çözümün ne kadar uzaklıkta beklenildiğine göre dinamik olarak değiştirilebilir.

2.2 Geleneksel Genetik Programlama

Genetik Programlama (GP) Evrimsel Algoritmalar'ın (EA) bir türüdür. EA doğru çözümü elde etmek için sınama-eleme (seçilim) ve türetme (çoğalım) adımlarını sürekli olarak takip ederek her turda (nesilde) öncekilere kıyasla, aranan değerlere daha uyumlu çözüm önerileri (denekler) elde etmeyi hedefler (Şekil 2.2). Başlangıçta rastgele yapılara sahip çok sayıda denek oluşturulur. Tüm denekler kullanıcı tarafından sağlanan bir fonksiyonla sınanır, arzu edilen sonuçla uyumluluklarını gösteren 0 ile 1 arasında bir skor alırlar. Görece yüksek puan alan denekler (uyumsuzlar) gruptan elenirler. Kalan denekler üzerinde çaprazlama ve mutasyon işlemleri uygulanarak grubun ortalama başarısını artıracağı beklenen yeni denekler elde edilir. Çaprazlama işlemini uygulamak

için gruptan denek çiftleri seçilir ve bunların farklı bölümleri birleştirilerek ikişer yeni denek elde edilir. Mutasyon işleminde az sayıda denek seçilir ve her birinin yapısında birer değişiklik yapılır. Bu işlem, uygulanmaması durumunda nesiller boyunca gerçekleşecek çaprazlamaların denek grubunu homojenize etmesinin ve sadece başlangıçta üretilen deneklerin rassallığının ulaştırabileceği çözümlerle sınırlı kalmanın önüne geçmek ve lokal minimum yerine global minimuma yaklaşabilmek için gereklidir.



Şekil 2.2 Evrimsel algoritmaların akışı

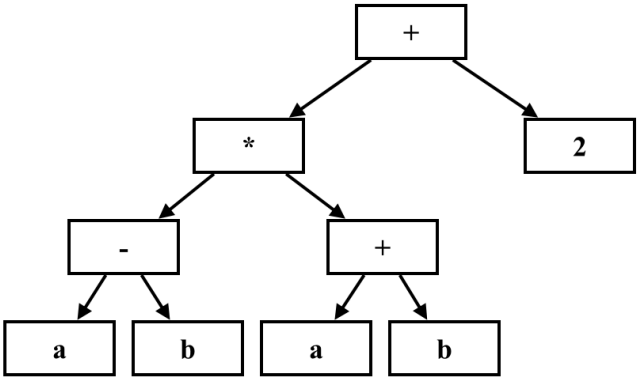
EA ilgili geçmiş çalışmalara bakıldığında bilgisayar programlarını ve matematiksel modelleri elde etmek için sıklıkla kullanıldıkları görülmektedir. Bu tür uygulamalarda deneklerin en kompleks halleriyle bile işlemci tarafından hızlıca test edilebilen fonksiyonlar olması, evrim sürecinin deneme-yanılma yapısından gelen aşırı işlem yükünü dengelediği için tercih edildiği düşünülebilir.

2.2.1 Deneklerin temsili

Yaygın olarak (Koza 1992a, Ferreira 2001, Poli vd. 2008) genetik programlamada ürün olarak sunulan “program”lardan kasıt matematiksel ifadelerdir (fonksiyonlardır). Erken çalışmalarda görülen programlar sabit değerlerin ve değişkenlerin aritmetik ve

mantıksal operasyonlarla işlenmesinden ibaret matematiksel ifadelerdir (ADF, GEP vs.). Bu ifadeler çoğunlukla TOPLAMA, ÇIKARMA, ÇARPMA, BÖLME, VE, VEYA, DEĞİL gibi işlemlerin ağaç yapısında bağlanmasıyla oluşturulur (ifade ağaçları). Alt seviyelerdeki işlemlerin sonuçları onları bağlayan işlemlere iletilir ve en sonda programın sonucu döndürülür.

Genetik programlama çözümlerinde kullanılan veri yapısı genellikle düğümleri aritmetik/mantıksal işlemlerden ve yaprakları parametrelerden oluşan bir ağaç veya onun lineer gösterimidir. Genetik programlama çalışmalarında sıklıkla karşılaşılan işlem ağaçlarının bir örneği Şekil 2.3'de gösterilmiştir.



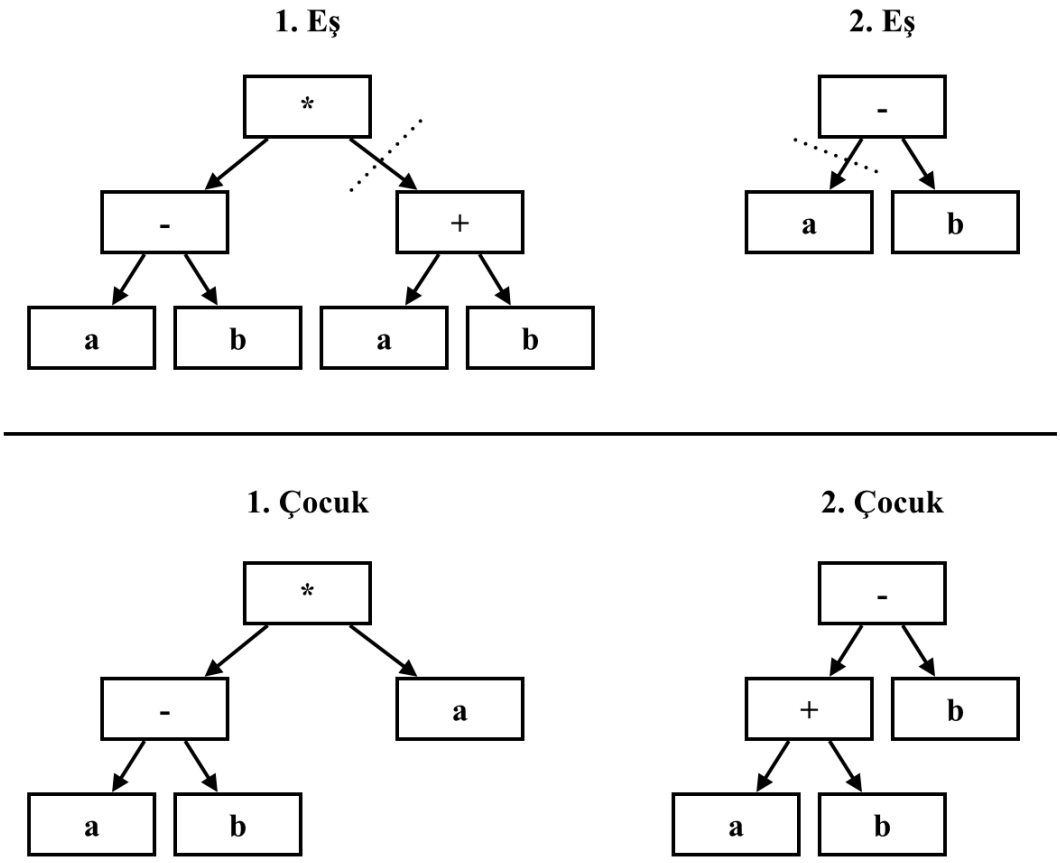
Şekil 2.3 (a - b) * (a + b) + 2 ifadesine denk gelen ifade ağacı

Sonraki çalışmalarda fonksiyonel programlama dilleri Lisp ve Haskell dilindeki fonksiyonların özelliklerinin eklenmesi denenmiştir. Bunlara bir örnek HOTGP dizilerin üzerinde işlem yapmayı sağlayan "reverse", "range" ve "filter" gibi yüksek-dereceli fonksiyonların çözüm tarafından kullanılmasına imkan tanır. Bu tür programlar güncel, zorunlu veya nesne yönelimli programalama dillerinin sunduğu olanakların çok azını kullanır. Güncel diller ek olarak değişken değer atamalarını, değer erişimini, akış kontrol araçları döngüler ve dallanmaları, özel veri yapıları tanımlarını, veri yapılarına özel yöntem tanımlarını, veri yapılarını birbirinden farkları üzerinden tanımlamayı ve diğer pek çok özelliği destekler. Evrim sürecini gerçek kaynak kodları üzerinden

yürütmek yerine 4 işlemle sınırlandırıp ifade ağacıyla temsil etmek çeşitli faydalar nedeniyle tercih edilmiştir. Bunlardan ilk göze çarpanı, ifade ağaçları üzerinde gerçekleştirilen mutasyon ve çaprazlama gibi genetik operasyonların geçersiz söz dizimine sahip programları ortaya çıkartmamasıdır. Mutasyonlarla gerçekleşen düğüm ekleme, çıkarma veya çaprazlamayla gerçekleşen alt-ağaç takasının sonuçları her zaman geçerli söz dizimine sahip ifadeler olmaktadır.

2.2.2 Çaprazlamalar

Genetik programlamanın pek çok varyasyonunda denekler arasında yani ifade ağaçları arasında çaprazlamalar gerçekleşir (Şekil 2.4). Gerçekleştirmek için iki ağaçtan birer düğüm seçilir ve bu düğümler alt ağaçlarıyla birlikte yer değiştirir.

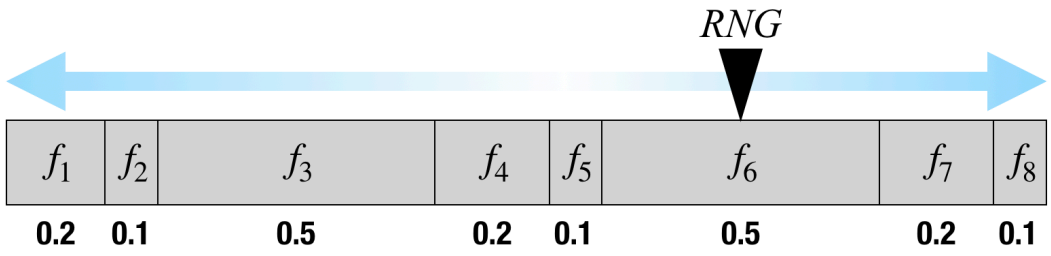


Şekil 2.4 İki atanın çaprazlamasının sonucu olan iki yavru

2.3.3 Eleme

Pek çok EA'nın kullandığı Elitist seçilim yöntemi n eleman arasından $n/2$ elemanı $O(n^2)$ zaman karmaşıklığı ile seçmesi nedeniyle tüm evrim sürecinin en uzun süren aşamasıdır. Bununla birlikte başarımı düşük ancak ilerlemeyi gömülü olarak ihtiva edebilecek denekleri erken elediği ve dolayısıyla çeşitliliği düşürdüğü için eleştirilir.

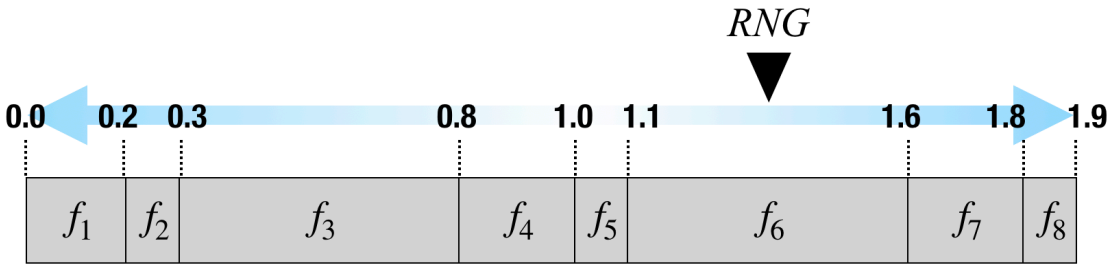
Havuzdaki çeşitliliği korumak için sıklıkla tercih edilen seçilim algoritmalarından biri başarım orantılı seçilimdir (Fitness Proportionate Selection, Rulet Çarkı). Bu algoritma havuzdan elenecek denekleri başarımlarıyla orantılı ihtimalle seçer. Yani başarımı 0.5 olan bir deneğin seçilme ihtimali 0.25 olan bir deneğin 2 katıdır. Şekil 2.5 bu algoritma için seçilebilecek en basit gerçeklemenin çalışma şeklini gösterir. Ancak bu gerçekleme n adet denek arasından $n/2$ adet deneğin seçimini $O(n^2)$ zaman karmaşıklığı ile sağladığı ve popülasyonlar (n) sıklıkla 1000'lerin üzerinde seçildiği için evrimsel algoritma problemlerinde tercih edilmez.



Şekil 2.5 Standart Rulet Çarkı algoritmasının çalışma mantığı

Şekil 2.6'deki Binary-Search düzenlemesi, basit versiyonun $O(n^2)$ ile gerçekleştirdiği seçimi $O(n * \log n)$ zaman karmaşıklığında gerçekleştirir. Binary Search'ün uygulanabilmesi için ihtimaller dizisinin cumulative (toplamsal) versiyonu kullanılır. Bu dizi bir kere oluşturulmasının ardından $(O(n))$ tüm seçimler için kullanılabildiği için seçim işleminin zaman karmaşıklığını etkilemez. Ancak bir elemanın birden fazla

seçimini engellemek için mükerrer seçimler elenir ve tekrar seçim yapılır. Mükerrerlerin elenmesiyle gerekli hale gelebilecek tekrar seçimleri azaltmak için gerekenin 2 katı kadar seçim yapılır.



Şekil 2.6 Binary Search ile kullanılabilir hale getirilmiş Rulet Çarkı algoritmasının çalışma mantığı

2.3 Genetik Programlamanın İlk Versiyonu ve ADF'ler

Koza (1992a) tarafından önerilen GP yönteminde çözümün tekrar kullanılabilen kod bloklarından oluşan şekilde bulunması hedeflenir. Bu kod bloklarının (fonksiyonlar) sayısı ve aldıkları parametrelerin sayıları sabittir ve başlamadan önce kararlaştırılır. Örneğin 10 (n) parametreli bir fonksiyonun edinimi hedefleniyorsa 2, 3, 4, 5, 6, 7, 8, 9 parametreli 8 adet ([2, $n-1$] aralığında) fonksiyon gövdesi oluşturulur.

Bu 8 fonksiyonu çağırabilen ilave bir fonksiyon ana programı temsilen eklenir. Kod blokları arasındaki hiyerarşi 1 kat ile sınırlanmıştır. Ana program tarafından çağrılabilen programlar bir diğerini çağıramazlar. Başlangıçta kullanıcı tarafından belirlenmesi gereken parametrelerden bazıları: terminallerin kümesi, fonksiyonların kümesi, başarım fonksiyonu, popülasyon sayısı, nesil sayısı, sonlandırma koşulu. Terminallerin kümesi çözümün girdi parametrelerinin kümesidir. Örneğin: İki girdili bir program için $\{x, y\}$. Fonksiyonların kümesi programın içerebileceği işlevlerin kümesidir. Örneğin $\{+, -, *, /\}$.

Başarım fonksiyonu her bir deneğin aranan çözüme uzaklığını ölçen fonksiyondur. Bu, çok sayıda durum için aranan çözüm ve eldeki çözüm arasındaki sapmayı hesaplayıp, toplam sapmayı döndüren bir fonksiyon olabilir. Süreç belirtilen nesil sayısına ulaşmadan, kabul edilebilir olarak görülen başarım değerine sahip bir denek ortaya çıktığında sonlanacak şekilde başlatılabilir. Başka bir çalışmada (Hooper vd. 1997) aktarılan değerlendirmeye göre Koza'nın seçtiği yüksek popülasyon genişliği çaprazlamanın yıkıcı etkisini azaltmaya yönelik etkiye sahiptir.

2.4 Tip Güvenliğini Gözeterek Arama Uzayını Daraltma

Birçok çalışmada dillerin bir özelliği olan "strong typing" özelliği, GP'nin arama alanını sınırlandırabilecek ve aramayı kısaltacak bir özellik olarak düşünülmüştür (Montana 1995, Banzhaf vd. 1998). Bu yöntemde (STGP); üretmesi gereken değer tipi (boolean, string, integer, float vs.) bağlanacağı üst düğümün kabul ettiği değerle belirli olan bir düğüm için, farklı tipte sonuç döndüren bir ağaç oluşturulmaz.

2.5 Asgari Değişiklik Gerektiren Sonucun Aranması ve N-Adımlı İyileştirme

Mutasyonların (olasılığa dayalı işleyişleri nedeniyle) bir yavruya sadece bir kere uygulanmasına karar verilmiştir. Çünkü bir mutasyonun n mutasyon arasından rastgele seçildiği düşünülürse bir uygulamada doğru mutasyonun seçilmiş olma ihtimali $1/n$ görülür[1]. Bir denek üzerinde seçilime girmeden art arda uygulanacak m mutasyonda tüm seçimlerin doğru gerçekleştirilmiş olma ihtimali $(1/n)^m$'dir. Dolayısıyla söylenebilir ki uygulama sayısı arttıkça hedefe ulaşma ihtimali düşer. Üstelik GP'nin süregelmiş en büyük sorunlarından biri olan şişmenin gözlenme ihtimalini yükseltir.

[1] Bu ihtimaller çözümün tek bir formu ve ona ulaşmanın tek yolu olduğu düşünüldüğünde geçerlidir ve mutasyon düğümünün seçimindeki ihtimali göz ardı eder. Gerçekte durum daha karmaşıktır ve ihtimaller daha düşüktür.

GP'de seçilim fenotiplerin başarım karşılaştırmalarına göre gerçekleşir ancak değişim ve ilerleme genotipte gerçekleşir. Fenotipte pozitif fark yaratacak bir değişim genotipte bazen birden çok üst üste değişim gerektirir.

Çalışmada anılan n-adımlı iyileştirmeler kısaca soyda bir davranışı ortaya çıkarmak için birden fazla genetik müdahale gereken iyileştirmeleri tanımlamak için kullanılmıştır. Biyolojik evrim alanında Kompleks Adaptasyon ifadesine de benzerlik gösterdiği görülmüştür (Wagner ve Altenberg 1996).

2.6 GP ile Kod Sentezi

GP ile bilgisayar programlarının bir geliştiriciye ihtiyaç duyulmadan elde edilebileceği 90'lardan bu yana yayınlanan pek çok çalışmada ve çeşitli yarışmalar sayesinde defalarca görülmüştür. Bu yarışmalardan biri olan ve 2004 yılından beri her yıl düzenlenen Human-Competitive yarışmasında GP'yi kullanan ve insanla yarışır düzeyde başarım göstermiş çalışmalara ödüller dağıtılmıştır (Human-Competitive 2023). Yapılan çalışmalar genetik programlamaya olan yüksek ilgiyi ve onun yüksek, rekabetçi potansiyelini gösterir niteliktedir. 30 yılı aşkın sürede araştırmacılar GP'yi çok çeşitli yönlerden değiştirerek geliştirmeye çalışmıştır. Bunlar deneklerin farklı yapıdaki temsilleri (tree, linear, cartesian vs.), çoklu genden oluşan temsiller (ADF [Koza 1992a], GEP [Ferreira 2002], HOTGP [Fernandes 2023] vs.), çeşitliliği koruyan seçilim algoritmaları (double-tournament vs.), deneği temsilden fenotip/genotip gibi özel yöntemlerle üretme (GEP, GenProg [Le Goues vd. 2012a], HOTGP vs.), evrim sürecini basitten-karmaşığa aşama aşama ilerletme (Layered Learning [Stone ve Veleso 2000]), test koşullarını popülasyonun altkümelerine paylama (Selection Architecture [Jackson 2009]), birim testlerini başarım fonksiyonu olarak kullanmadır (Le Goues vd. 2012a).

2.7 Zorunlu Dillerde GP

Zorunlu diller programdan beklenen davranışı yani girdi parametrelerine göre döndürmesi gereken değerleri oluşturmak için ortaya koymak için (özellikle fonksiyonel dillerden farklı olarak) değişkenlerle erişilen hafıza bölgelerindeki sayısal değerlere erişmek ve onları güncellemek için özel kontrol akışları oluşturmaya imkan tanır. Kontrol akışı kurgulamada geliştiriciye sunulan araçlar çoğunlukla döngüler ve dallanmalardır. Tıpkı fonksiyonel diller gibi özel kontrol akışları genellikle fonksiyonlar denilen deyim listelerinden oluşan yapılar içinde oluşturulur ve kodun farklı bölgelerinden çağrılabilirler. Zorunlu dillerde sıklıkla karşılaşılan bir özellik fonksiyonların fonksiyon dışında tanımlanan değişkenlere ve dosyalara erişebilmesidir. Dolayısıyla zorunlu dillerdeki fonksiyonlar yan etkilere sahip olabilir. Bu özellik de, bir fonksiyonun parametrelerinin aynı değerlerdeki farklı çağrımları için farklı sonuçlar üretebileceği anlamına gelir. Çıktılar doğrudan girdilerin sonucu olmak zorunda değildir.

Zorunlu dillerin zengin özellik setleri onları GP'ye uyarlamada daha zor kılar. Çünkü mutasyonlar sırasında deneğe eklenebilecek her bir deyim artık daha geniş bir set içinden seçilir. Ayrıca, fonksiyon içindeki bir deyimin ürünü/sonucu direkt olarak bir sonraki deyim tarafından alınmak yani ardışık deyimlerin girdileri ve çıktıları zincir biçiminde bir sonrakine bağlı olmak zorunda değildir. Deyimlerin fonksiyon gövdesi içindeki dizilimlerindeki kıstaslar değişken tanımlarının (ve ilk değer atamalarının) erişimlerinden önce yer alması gibi daha gevşek kurallardır. Bu durum bir alttaki satırın görevinin daha geniş bir set içinden belirlenebileceğini gösterir. Dolayısıyla fonksiyonel dillere göre doğru deyim dizilimi için daha fazla aday olmasıyla birlikte genel olarak çok daha fazla anlamlı-anlamsız deyim kombinasyonu bulunur (Şekil 2.7).

```python
sayac = 0
for i in range(5):
    sayac += 1
    if sayac % 2 == 0:
        print(f"{sayac} çift sayıdır")
    else:
        print(f"{sayac} tek sayıdır")

# 1 tek sayıdır
# 2 çift sayıdır
# 3 tek sayıdır
# 4 çift sayıdır
# 5 tek sayıdır

def greet(name):
    return f"Merhaba {name}!"

ileti = greet("Dünya")
print(ileti) # Merhaba Dünya!

```

Şekil 2.7 Zorunlu dillerin imkanlarını kullanan bir Python betiği

Bu bağlamda zorunlu dillerde Genetik Programlama fonksiyonel dillerdeki GP uygulamalarından farklı mücadeleleri ve imkanları barındırır.

Klasik genetik programlamadaki denek/aday programlar yapısal olarak her zaman geçerli ancak ürettikleri sonuçlar aranılanlarla değişken uyum gösteren ifade ağaçları ve S-İfadeleridir. Fonksiyonel programlamadaki GP uygulamalarından denek üretimi ve mutasyonu sırasında tip uyumu gözeten yöntemlerin ürettiği denekler aynı şekilde her zaman geçerli yapıya sahip olan ancak başarımı farklı olabilen deneklerdir. Zorunlu dillerde bunlara ek olarak deyimlerin, değişkenlerin tanımlandıktan önce erişilmeleri gibi farklı sebeplerle yanlış sırada dizilmelerinden kaynaklı hataya sahip deneklerin üretilmesi öngörülebilir.

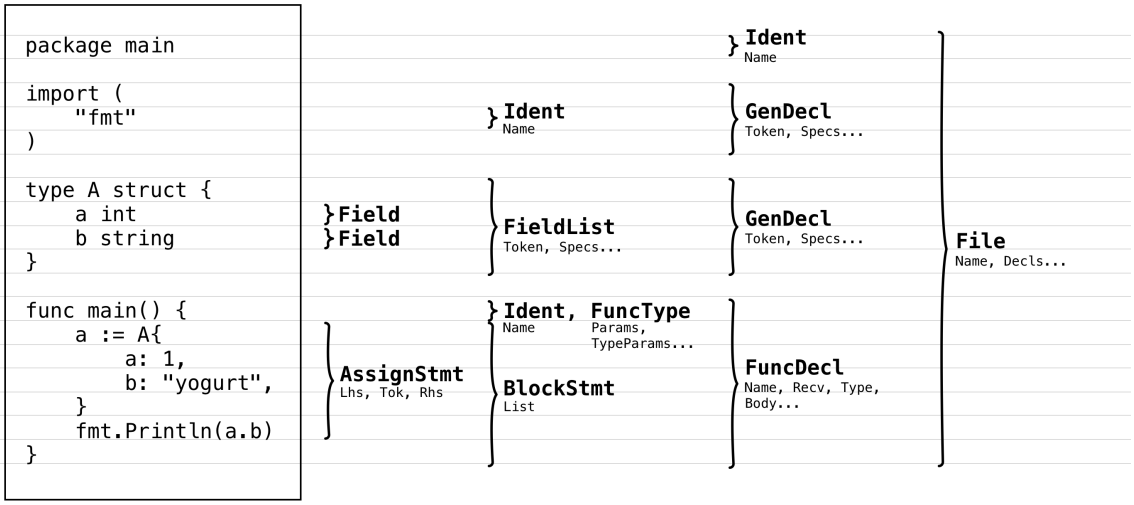
Şekil 2.8 zorunlu yapılı bir dil olan Go dilinde, genetik programlamanın konusu olabilecek kelime test çevirme fonksiyonunun sentezi sorununda ortaya çıkması beklenebilecek denekleri göstermiştir. Bu örneklerin hiçbiri sözdizimi hatalarına sahip değilken sadece sonuncusu semantik olarak istenene uygundur.

```go
func WordReverse(s string) string {
    return ""
}

func WordReverse(s string) string {
    return "dlrow olleH"
}

func WordReverse(s string) string {
    return s
}

func WordReverse(s string) string {
    reversed := ""
    for i := 0; i < len(s); i++ {
        reversed += s[i:i]
    }
    return reversed
}

func WordReverse(s string) string {
    reversedBytes := []byte(s)
    for i, j := 0, 0; i < j; i = i + 1 {
        reversedBytes[i], reversedBytes[j] = reversedBytes[j], reversedBytes[i]
    }
    return string(reversedBytes)
}

func WordReverse(s string) string {
    reversedBytes := []byte(s)
    for i, j := 0, len(reversedBytes)-1; i < j; i, j = i+1, j-1 {
        reversedBytes[i], reversedBytes[j] = reversedBytes[j], reversedBytes[i]
    }
    return string(reversedBytes)
}
```

Şekil 2.8 Zorunlu yapıdaki bir dildeki kodun sentezi sırasında oluşması beklenebilecek farklı başarımlara sahip aday çözümler

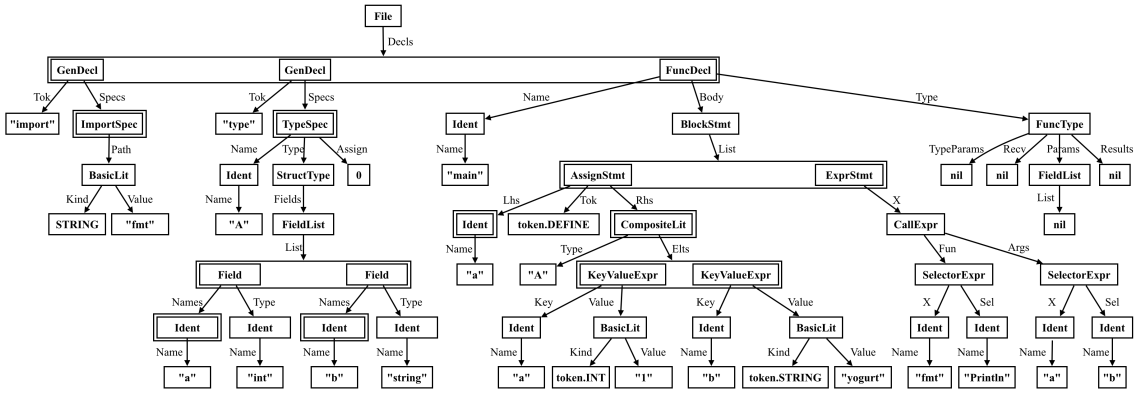
2.8 AST

AST, geliştiriciler tarafından yazılan kaynak kodlarının derlenmeden önce derleyici tarafından dönüştürüldüğü bir temsildir. Bir AST, bir kaynak kodu içindeki tüm deyimleri, belirtimleri ve bildirimleri koddaki hiyerarşileri içinde tutar. Bir düğümün tüm astları kodda da içinde yer alan yapıları tutar. Örneğin koddaki bir If bloğunun koşul kısmına yazılan ifade AST içinde bir If deyimini temsil eden düğümün koşul alanına atanmış ikili türde deyim, çağrı deyimi vs. ile temsil edilir. Şekil 2.9'da örnek bir Go kodu için resmi ayrıştırıcı tarafından üretilmiş AST bazı düğümleriyle gösterilmiştir. Ağacın tamamı deyim, belirtim ve bildirim tiplerinde 100'ün üzerinde düğümden oluşmaktadır (Şekil 2.10).



Şekil 2.9 Örnek bir Go koduna ait AST'nin bazı düğümleri

Şekil 2.10'daki temsilde her bir veri yapısının bileşenlerinin isimleri okların üzerinde etiketlenmiştir. Oklar, işlevi koddaki daha küçük yapıları temsil etmek olan değerlere veya bunların dizisine ulaşır. Oklar bütünlerden parçalara işaret eder. İşaret edilenler bazen doğrudan daha küçük parçalar, bazen parçalardan oluşan listeler bazen de ilkel tipte değerlerdir.



Şekil 2.10 Bir kod örneği için go/parser (v1.20) tarafından üretilmiş AST

2.9 ASTGP

ASTGP, program varyantlarının temsili için derleyiciler için tasarlanmış olan AST'lerin kullanıldığı GP uygulamalarıdır. Genetik operasyonlar AST üzerinde gerçekleştirilirken programların oluşturulması için AST koda dönüştürülür (ve derleyiciye veya yorumlayıcıya verilir).

Düzenlemeleri kod yerine AST üzerinde yapmak sözdizimsel hataların bir kısmının oluşumunu engeller. Örneğin bir anahtar kelime (func) bir özel isim ve girdi ile çıktı parametre listesinden oluşan fonksiyon tanımını kodda oluşturmak için gereken karakter kombinasyonu çok fazlayken, aynı sonucu AST ile üretmek için tüm düğüm tipleri arasından önce fonksiyon tanımlarını temsil eden düğümün seçilmesi ($p = 1/56$) yeterlidir. Kodda fonksiyon tanımının parçası olan parametre listeleri AST düğümünün birer alanıdır.

ASTGP çoğunlukla modern dillerde kullanılmaktadır. Bu dillerin bir kısmı da zorunlu yapıdadır. Koza tarzı ifade ağaçlarının sıklıkla kullanıldığı fonksiyonel dillerin aksine, zorunlu dillerde kod doğrudan girdi değerleri çıktı değerlere dönüştürecek aritmetik ve mantıksal operasyonlardan ibaret değildir. Asıl olarak kod kontrol akışını tanımlar ve

değişkenlerde saklanan değerleri düzenler. Genotipteki ve fenotipteki farklılaşma zorunlu dillerin AST'lerinde fonksiyonel dillere göre biraz daha belirgindir.

2.10 Birim Testi

Bir fonksiyon veya veri yapısı gibi bir kod parçasının (Şekil 2.11) istenildiği gibi çalıştığını onaylamak için hazırlanan ve her değişikliğin ardından tekrar çalıştırılan fonksiyonlara birim testi (Şekil 2.12) denir. Bu testler hedef birimin kabul etmesi ve değer üretmesi gereken her çeşit girdi değerini farklı çağrılarda göndererek geri dönen değerleri beklenenlerle karşılaştırır. Hiçbir değer karşılaştırmasında beklenen ve elde edilen arasında değer eşleşmezliği yaşanmayan birim testi geçer.

```
1   package search
2
3   func Index(s []string, k string) int {
4       for i, v := range s {
5           if v == k {
6               return i
7           }
8       }
9       return -1
10  }
11
```

Şekil 2.11 Bir birim testinin konusu olan fonksiyon

80 binin üzerinde kullanıcının katıldığı Stackoverflow.com (2023) anketinde "Organizasyondaki süreçler, araçlar ve programlar" sorusuna yanıt veren yaklaşık 42 bin kullanıcının seçimlerinde görülmektedir ki yazılım geliştirilen organizasyonların çoğunluğunda otomatik test mekanizmaları kuruludur.

```
run package tests | run file tests
1  package search
2
3  import "testing"
4
   run test | debug test
5  func TestIndex(t *testing.T) {
6      type testcase struct {
7          inputSlice []string
8          inputKey   string
9          want       int
10     }
11
12     tcs := []testcase{
13         {
14             inputSlice: []string{"elma", "portakal", "çilek"},
15             inputKey:   "portakal",
16             want:       1,
17         },
18         {
19             inputSlice: []string{"yoğurt", "su"},
20             inputKey:   "portakal",
21             want:       -1,
22         },
23         {
24             inputSlice: []string{},
25             inputKey:   "portakal",
26             want:       -1,
27         },
28     }
29
30     for _, tc := range tcs {
31         got := Index(tc.inputSlice, tc.inputKey)
32         if got != tc.want {
33             t.Fatalf("want %q, got %q\n", tc.want, got)
34         }
35     }
36 }
37
```

Şekil 2.12 Bir sonuç karşılaştırması ve üç test senaryosu içeren bir birim testi

2.11 Test-Güdümlü Geliştirme

Yazılımlar geliştirici organizasyonlar için sözleşmeler ve kanunlardan kaynaklı olarak yükümlülükler getirmektedir. Bununla birlikte, yazılımın sunduğu kullanıcı deneyimi ve kullanıcıların beklentilerinin tatmini organizasyonun daha fazla müşteriye ulaşmasını etkiler. Tersi bir durumda örneğin kullanıcı deneyimini veri veya zaman kaybı gibi olumsuz etkileyebilecek veya veri sızıntısıyla şahıs güvenliğini tehlikeye atacak hatalı kod içeren bir sürümün dağıtımı organizasyonlara doğrudan veya uzun vadede ciddi maddi zararlar getirebilmektedir (Anonymous 2022). Geliştiriciler bu yüzden yazılımsal ürünlere yeni özellik ekleme amacıyla kod üzerinde düzenlemeler yaparken daha önceden yazılan ve mevcutta çalışmaya devam eden işlevleri bozmamaya özen gösterirler.

Yazılım geliştiriciler şirketteki kod tabanının uyumsuz gelişimini tespit edebilmek için geçmişte geliştirilmiş kodların değişmekte olan gereksinimlere de uygun olduğunu sürekli olarak kontrol etmelidirler. Bu kontrollerin ardından bazen düzenlenmesi gereken birimler saptanır. Kod parçasının yerine getirdiği ancak belgelenmemiş işlevlerinin var olma ihtimali geliştiricilerin özgüvenini düşürür. Geliştiricilerin fazladan zaman harcamasına neden olur. Bu durumları engellemek için Test Güdümlü Geliştirme (Test-Driven Development [TDD]) yaklaşımı ortaya çıkmıştır.

Test güdümlü geliştirme geliştiriciler tarafından, geliştirme sürecini testler üzerinden sürdürerek gerçekleştirilir. Bu testler farklı türlerde olabilir. En yaygın kullanılanlar entegrasyon ve birim testleridir. Birim testleri genellikle fonksiyonlar ve metotlar için yazılır. Entegrasyon testleri ise proses gibi yürütülebilirlerinin parçaları olduğu daha büyük bütünleri test eder. TDD yaklaşımında gerçekleme aşamasına geçilmeden önce ve sistem parçalara ayrıldıktan sonra, kodda yer alacak birimler belirlendikten sonra her bir birimin testleri yazılır. Bu testlere birim testi denir. Sürekli-Entegrasyon/Sürekli-Konuşlandırma (CI/CD) döngüsünün her tetiklenişinde kod tabanının son hali, birim-testlerinin tamamıyla tekrar test edilir. Birim testleri genellikle Düzenleme-Eylem-

Karşılaştırma adımlarından oluşan bir formdadır. Düzenleme adımında hedef birimin sınanması için gerekli olan girdi parametreleri hazırlanır. Eylem adımında girdi parametreleri hedef birime gönderilir ve çıktı parametreleri saklanır. Karşılaştırma adımında girdi parametreleri için beklenen çıktı parametreleriyle eylem adımında elde edilen çıktılar karşılaştırılır. Karşılaştırmada eşleşmeyen sonuç bulunması durumunda birim-testi başarısızlıkla sonuçlanır. Birim testleri genellikle örnek girdi ve bu girdiler için beklenen çıktıların bir listesini içerir ve bir döngü ile hedef birim her bir çift ile teker teker test edilir. Bazı birim-testleri birimin hata vermesi gereken durumları da özellikle test eder.

Tüm birim-testlerini geçen kod tabanı CI/CD içinde bir sonraki bölüme ilerler. Aksi durumda ise döngü durur ve geliştiriciler sorundan haberdar edilir. Hatalı kodun sevkiyatı engellenerek kullanıcı memnuniyetinin ve güvenliğinin zedelenme ihtimali düşürülür.

Yazılım geliştiren organizasyonlarda web geliştirmeyle ilgili olan REST API desenini takip eden programlarla sıklıkla karşılaşılmaktadır. Bu tür programlar genellikle bir HTTP isteğini okur, işler ve ona yanıt gönderir. Bu programların diğer türlerdeki programlardan farkı, yapılan işlerin çoğunlukla metinsel veya sayısal kaynakların oluşturulması, erişilmesi, silinmesi ve güncellenmesi (CRUD) temelli olmasıdır. Dolayısıyla GP'nin REST API projelerinde kullanılabilecek olgunluğa erişmesi bu projelerin yazılım geliştiren organizasyonların iş yükü ve kazancındaki yüksek payı, teknik gereksinimlerinin sınırlılığı ve görece tekdüzeliği, bu organizasyonların zaten TDD ve CI/CD nedeniyle birim-test hazırlama kültürünü benimsemiş olması nedeniyle Genetik Programlamanın akademik çevrelerden ticari organizasyonlara doğru ilerleyişi ve önem kazanmışında önemli adımlardan biri olması beklenmektedir.

Choma vd. (2018) tarafından yayınlanan çalışmada ortalama tecrübeleri 10 yıl olan 19 yazılım geliştiriciyle yapılan bir anketin sonuçları paylaşılmıştır. Ankete göre geliştiricilerin gözünde birim-testlerinin faydaları daha yalın kod yazmayı sağlaması,

kod düzenlenmelerinde özgüveni artırması ve hataları önlemeyi kolaylaştırmasıdır. Ayrıca katılımcıların %42'sinin yazılım tasarım sürecini TDD ile yürüttükleri, %32'sinin ise zaten tanımlanmış sınıflar ve metotlar için gerçeklemeden önce birim-testlerini geliştirdiği aktarılmıştır.

3. GEÇMİŞ ÇALIŞMALAR

GP şu ana kadar pek çok çalışmada matematiksel ifadelerin üretimi amacıyla farklı dillere uyarlanmıştır. Bunlardan ikisi Lisp (Koza 1992b, Ferreira 2002) ve Python'dır (Sipper 2019). GP'nin matematiksel ifadelerin eldesi dışında kod (program) sentezi amacıyla da kullanılmaya başlandığı görülmüştür. Bu denemelerden bazıları C (Weimer vd. 2009, Le Goues vd. 2021), Java (Yuan ve Banzhaf 2017, Le vd. 2016) ve Haskell (Fernandes vd. 2023) dillerinde kod üretimini hedeflemiştir.

GP ile kod sentezi hedefleyen çalışmalar uyarlama sırasında o dilde karşılaşılan sorunları da tanımlamıştır. Örneğin derlemeli dillerdeki uzun çalışma süresi, zorunlu dillerdeki yan etkilere (side-effect) sahip olabilen fonksiyonların test ortamının deterministiki yapısını bozması, nesne yönelimli dillerde ifade ağacına dahil olan deyimlerin arama uzayını genişletmesi.

GP'yi bir kod/program sentezi çözümü olarak sunabilmek için yazılım geliştirilen organizasyonlarda sıklıkla kullanılan modern dillere de uyarlamak gerekmektedir. Bir geliştirici platformu anketine göre bu dillerden en yaygın kullanılanları JavaScript, Python, TypeScript, Java, C#, C++, C, PHP, Go ve Rust'tır (Stackoverflow 2023). Python, TypeScript ve Go son yıllarda bu anketteki yükselişiyle dikkat çekmektedir (Yıldırım 2023).

3.1 GP ile Matematiksel İfadelerin Üretimi

Rosca ve Ballard (1994) evrim süresince ortaya çıkan deneklerin bünyesinde sık tekrar eden "sık rastlanan bloklar" veya özel nitelikleri karşılayan "uyumlu bloklar" denilen işlev bloklarını ortaya çıkarmayı hedeflemiştir. Bu blokları fonksiyon haline getirerek genetik operasyonların deneğe ekleyebileceği (denek içinden çağrılabileceği) fonksiyonların arasına eklenirler. Bu sayede kodun tekrar kullanımı hedeflenir. Paylaşılana göre, sık rastlanan bloklar kod için vazgeçilmez değillerdir. Ayrıca pek çoğu

erken nesillerde ortaya çıkıp, denek grubunu hızla domine edip, kısa sürede ağırlıklarını kaybetmektedirler. Uyumlu blokları oluşturmak için denek bünyesindeki işlev blokları uyumluluk skorlarıyla incelenir. Bu inceleme için 3 yol vardır, ana başarım fonksiyonu alt görevleri test edebilir; başarım fonksiyonu ana çözümün daha az boyutunu test edecek şekilde değiştirilebilir, sürecin ilerleyen evrelerinde bu boyutlar kademeli artırılır ta ki tümüne ulaşıncaya kadar; kullanıcıdan birden çok başarım fonksiyonu alınabilir.

Hooper vd. (1997) tarafından yayınlanan makalede overfitting, çeşitliliği kaybetme, yerel-minimuma takılma ve seçilim evresinin uzun sürmesi sorunlarına çözüm aranmıştır.

Luke ve Panait (2002) tarafından yayınlanan makalede GP'nin ana sorunlarından olan şişmeye karşı mevcuttaki önlemler incelenmiş ve ayrıca yeni bir yöntem aranmıştır. Anılan yöntemler: derinlik limiti (maximum depth restriction), atalarından iyi olmayan yavruların elenmesi (pseudo-hillclimbing).

Koza (2010), Human-Competitive yarışmasına katılmış ve ödül kazanmış GP araştırmalarını incelemiştir. Bu çalışmaların bazılarının patentlendiği görülmüştür. Çalışmaların alanları bilgisayar devreleri, mekanik/optik sistemler, otomatalar olarak aktarılmıştır. Bunun yanında listelenen çalışmalar arasında program tamiri ve cebir gibi konulardaki çalışmalar da görülmüştür. Listelenen 78 çalışmada ismiyle en çok karşılaşılan araştırmacılar John R. Koza (40), Martin A. Keane (30), Forrest H Bennett III (22), David Andre (16), Matthew J. Streeter (10) olmuştur.

3.2 GP ile Fonksiyonel Dillerde Çözüm Üretimi

Jackson (2009) tarafından yayınlanan bir çalışmada hedef fonksiyonun sonuç döndürmesi gereken girdi parametrelerinin değer aralıkları tüm deneklere uygulanmamış, bunun yerine aralıkların parçaları denek gruplarına paylanmıştır.

Selection Architecture (SA) olarak adlandırdıkları bu yöntemde popülasyondaki denekler gruplara ayrılır. Her bir grubun başarım fonksiyonu kendine özeldir ve bir gruptaki denek başka bir gruptaki deneğin sağladığı kriterleri sağlamak zorunda değildir. Hedef çözümün karşılaması gereken koşullar ayrılarak gruplara dağıtılır. Örneğin bir sembolik regresyon uygulamasında arzu edilen aralık [0, 1) ise ve 2 grup kullanılacaksa, bir gruptaki denekler yakınsanacak fonksiyonun [0, 0.5) aralığındaki x değerleriyle, diğer gruptaki denekler ise [0.5, 1) aralığındaki değerleriyle test edilir. Payların yeniden düzenlenmesiyle sonuçların elde edilmesini hızlandırmanın mümkün olduğu görülmüştür. SA yöntemi ile gerçekleştirilen even-5-parity öğrenimi uygulaması sonucunda elde edilen sonuçlar geleneksel GA ve ADF yöntemleri kullanılarak elde edilen sonuçlarla karşılaştırılmış ve 8 dal kullanılan SA yönteminin ADF'nin 3 katı uyumlu sonucu ürettiği ve 54'te 1'i kadar işlem gerektirdiği görülmüş. Testlerin farklı dallara pay edilmesi, elde edilen hiçbir sonuçların hiçbirinin tüm testleri karşılamamasıyla sonuçlanmış. Bununla birlikte bu yaklaşımın, problemin çözümünde kritik rolü olabilecek değer aralıklarının sadece bazı dallara gösterilmesi ve diğer dallardaki evrim sürecini bu dallara düşen bu kritik bilgilerden mahrum bıraktığı görülmüştür.

3.3 ASTGP Uyarlamaları

Yuan ve Banzhaf (2017) tarafından yayınlanan çalışmada Multi-Objective GP Java kodunun tamiri için kullanılmıştır.

Illanes ve Bergel (2021) çalışmalarında GP'yi Nesne Yönelimli Programlamaya uyarlamıştır. Çalışma AST'nin, birim testlerinin ve Levenshtein uzaklığının kullanımı ile dikkat çekmektedir. Araştırmacılar iki denek arasındaki yapısal benzerliği saptamak için kodları arasındaki Levenshtein uzaklığını esas almışlardır. En popüler 10 dilden 9'unun nesne yönelimli, 6'sınınsa dinamik tipli olması çalışmaya göre GP'nin bu özellikleri dillere uyarlanma çalışmalarını önemli kılmaktadır. Yazarlar koddaki bir satırlık

eksiklikleri %51 hassasiyetle bulabildiklerini paylaşmışlardır. Yöntem Pharo programlama dilinde gerçeklenmiştir.

Pantridge vd. (2022) tarafından paylaşılan çalışmada tanıtılan CBGP (Code Building GP) AST kullanımı ve sadece tip güvenliği sağlayan denekler üzerinden ilerlemesiyle öne çıkmaktadır.

Fernandes vd. (2023) tarafından yayınlanan çalışmada HOTGP adında bir GP yöntemi tanıtılmıştır. Yazar GP'de yaygın olarak görülen ve iyileştirme için seçtiği 3 sorunu aktarmıştır. Bu sorunlar; kodda geçen değerlerin herhangi birinin üzerindeki en küçük değişimin bile başarım değerini tamamen etkilemesi, GP'de başarıya ulaşmanın, kullanıcının, sağladığı testte gerekli bütün durumları hesaba katmasına bağlı olması ve durumsal fonksiyonları test etmenin bu fonksiyonların sonuçları öngörülemeyebileceği nedeniyle GP'yi yanıltmasıdır. Yazar bu sorunlara çözüm olarak arama uzayını pure fonksiyonlardan ibaret kılmış ve STGP'yi takip etmiştir. Pure fonksiyonlar yerlerine çıktıları konulduğunda gerisinde hiçbir şeyi değiştirmeyen, yan-etkileri olmayan fonksiyonlardır. Çözüm lamda fonksiyonları (değişkenlere atanan fonksiyonlardır), yüksek dereceli fonksiyonları (giriş/çıkış parametrelerinden biri başka bir fonksiyon olan fonksiyonlardır) ve parametrik çok biçimliliği destekler. Haskell diline uyarlanmıştır. Lamda fonksiyonlar nedeniyle ürettiği çözüm alt-programlar içeren yapıdadır. Ağaç derinliği ana fonksiyon için 15, lamdalar için 3'tür. PSB2 bazlı pek çok kıyaslamada sınanmıştır.

3.3.1 Program tamiri

Weimer vd. (2009) tarafından C dilinde denenmiştir. Yöntem, bir dizi pozitif ve negatif test durumu içeren birim testlerini ve mevcutta kısmen çalışan bir gerçeklemeyi girdi olarak alır ve evrimsel arayış ile kod yapısında minimum değişiklikle hatayı gidermeye çalışır. Genetik operasyonlar (mutasyon ve çaprazlama) kod üzerinde değil AST üzerinde yapılır. Bu sayede sözdizimi hataları (eşleşmeyen parantezler vs.) içeren

deneklerle zaman kaybetmenin önüne geçilir. Girdi gerçeklemenin halihazırda sahip olduğu işlevleri bozmamak için, genetik operasyonların uygulanacağı bölgeleri seçerken pozitif sınama durumlarının yürütümü sırasında gezilen deyimlere karşın, negatif durumların testlerinin yürütümü sırasında gezilen deyimleri öncelikli tutar. Arama uzayını daraltmak için çözümü minimum değişiklikle bulmayı hedefler. Gerçeklemeyi sıfırdan elde etmek hedeflenmemiştir. Bu nedenlerden dolayı tam bir genetik programlama uygulaması olarak görmek zordur.

Le Goues vd. (2012a) tarafından tanıtılan GenProg (Genetic Program Repair), hatalı C kodu için gerekli yamanın kod için yazılmış testler ile bulunmasını hedefler. GenProg kod içindeki deyimler üzerinde çeşitli işlemler uygulayarak hatalı kodun farklı versiyonlarını üretir. Bu işlemler: yer değiştirme, silme, çoğaltma. Çoğu GP'nin aksine arama-uzayı üzerindeki açılmasını sınırlı tutar, en az sayıda kod modifikasyonuyla sonuç üretmeye odaklanır (Repair Minimization). Bunun için 3 yöntem uygular. a) AST'yi sadece deyim seviyesinde inceler (daha küçük ve daha büyük yapılarla örneğin ifadelerle ilgilenmez). b) Hiçbir zaman sıfırdan kod oluşturmaz, hatasız kısımlardan çoğaltır ve taşır. c) Genetik operasyonların etki edeceği kod bölgesini, hata alınan testte yürütülen bölgeyle sınırlandırır. Bu önlemlerle arama uzayı çoğu durumda çözümü mümkün kılacak kadar daraltılmış olunur. Encoding, her bir adayın denk geldiği kodu tutmaz. Bunun yerine ilk koddan o deneğin son halini üretecek modifikasyonların listesini tutar. Yazara göre bu kodlama yöntemi çözümün ölçeklenmesini kolaylaştırır. Başarım ölçümleri için süre aşımını da kontrol eden VM veya korumalı alan (sandbox) önerilmiştir. Derlenemeyen örneklerin başarımı 0 kabul edilmiştir. Çözümün sınandığı sorunlarda ortalama tamir süresi yaklaşık 6 dakika olarak bulunmuştur. Bu sürenin önemli bir kısmı test ortamının hazırlanmasıdır.

Le Goues vd. (2012b) çalışmasında GenProg'u cloud üzerinde konuşlandırarak çalışma süresini kısaltmış ve büyük ölçekli kod tabanlarındaki uygulanabilirliğini artırmış. 105 sınamayla yapılan deneylere göre bir çözümün üretim süresi 1 dakika ve maliyeti yaklaşık $8 bulunmuştur.

3.4 Şişme ile Mücadele

Şişme (bloat) sorunu, genetik operasyonlar sonucunda oluşan ve yapılarında gerekenden çok daha fazla düğüm olan büyük boyutlu yavruların nesiller sonunda popülasyondaki çoğunluğu ele geçirmesi ve çözüm adayı olabilecek ifadelerin ortaya çıkmasını zorlaştırmasıdır (Poli vd. 2008). Bu tür yavrular asimetrik çaprazlamaların ve mutasyonların sonucu olabilirler. GP'nin tanıtımından bu yana şişme sorununa pek çok çözüm önerisi getirilmiştir. Koza (1992) deneklerin sahip olabileceği maksimum derinliği sınırlamıştır (depth-limiting). Hill-Climbing yönteminde seçilen bir atadan rastgele seçilmiş genleri çıkartarak oluşturulan yavru, başarım değeri atasından daha düşük olmadığında atasının yerine geçer. Bu sayede gereksiz hantallaşmanın önüne geçilir. Hooper vd. (1997) Recombinative Hill-Climbing (RHC) adını verdikleri yöntemde diğer Hill-Climbing yöntemlerinin aksine yerel-minimaya takılma ihtimali daha düşük olduğu için ata deneği rastgele seçmişlerdir. Pantridge vd. (2022) çalışmalarında en iyi deneği ata seçerek Hill-Climbing uygulamıştır. Bu tür yöntemler sonuca etkisi olmayan parçaların çıkarılmasını içerdiği için Generalization veya Simplification olarak da anılır.

3.5 Erken Elenme

İlerlemeyi korumak, ileriki yinelemelerde başarım değerine yansıyacak bir ilerlemeyi ortaya çıkarma ihtimali olan ancak mevcutta vasat çoğunluktan daha iyi başarıma sahip olmayan denekleri, bu gelişimi somut olarak ortaya koyabilmesi için ihtiyaç duyulan süreden önce elememeyi gerektirir. Stanley ve Miikkulainen (2002) NEAT çalışmasında sorunu çözmek için önlemler almış. Gelişme sürecindeki kısımların başarımı etkilememesi için genlerin pasif olarak bulunabilmesine olanak tanınmıştır. Ayrıca soy takibini uygulayarak deneklere elenmeden önce gelişimlerine devam etmeleri için zaman tanımıştır.

Konu "Premature elimination" ifadesiyle yakın dönem makalelerde anılmaya başlanmıştır (Agapitos vd. 2017, Grudniewski ve Sobey 2019).

3.6 Türleştirme

Stanley ve Miikkulainen (2002) tarafından tanıtılan NEAT (NeuroEvolution of Augmented Topologies), değişken yapıda yapay sinir ağlarını (Artificial Neural Network - ANN) bulmak için evrimsel süreçlerden yararlanan bir genetik algoritmadır. Genetik ilerlemeyi (kazanımı) korumak için soy takibi yapar. Optimum sonucu en verimli topolojiyle bulmayı hedefler. Farklı topolojideki ağların çaprazlanması ile genetik çeşitlilik sağlanır. Aynı yöntemi izleyen alternatiflerinin karşılaştığı sorunlarda iyileştirmeler hedefler. Bu sorunlar: prematüre eleme (premature elimination), şişme (bloat), tamamen farklı topolojilere sahip iki alternatif arasında çaprazlama. Araştırmacılar ayrıca denek bütünündeki parçaların farklı soydan alternatiflerinin yanlış noktadan çaprazlama sonucunda aynı deneğe toplanabildiğini ve bu yüzden başarımı daha düşük nesiller üreyebildiğini (Competing Conventions) aktarmıştır. Neuroevolution bir yapay sinir ağını topolojisiyle, kurallarıyla ve parametreleriyle birlikte elde etmeyi sağlayan evrimsel algoritma türüdür. Sinir ağını Gradient Descent kullanmaksızın ortaya çıkarır. Bu yüzden veri setinin yönlendirdiği yere ulaşmaz. Bunun yerine, aranan gereksinimleri karşılayan (ve veri setiyle uyum sağlayan) rassal denekler üzerinden başarım ölçütünde global minimuma ulaşmayı hedefler.

4. ARAÇLAR

Çalışmada faydalınan araçlar Go diliyle birlikte sunulan statik analiz araçlarıdır. Bu araçların denklikleri pek çok programlama diliyle birlikte sunulduğu için tanıtılan yöntemin farklı dillere uyarlanabilirliği yüksektir.

4.1 Go Derleyicisi

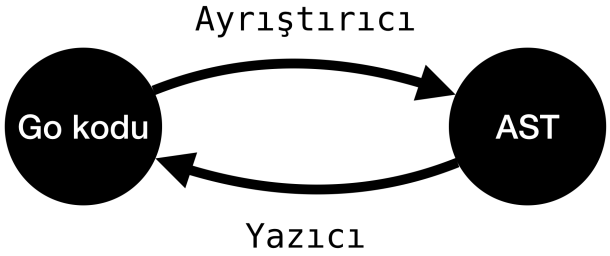
Paketler halinde düzenlenmiş Go kodunu çalıştırılabilir dosyalara dönüştüren bir programdır. Her bir deneğin testi sırasında bu programdan yararlanılır. Program olmayan deneklerin derlenmesi sırasında derleyici tarafından gösterilen sözdizimi hataları başarım değeri hesaplanması için kullanılmıştır.

4.2 Go Statik Analiz Araçları

Go kodu ve AST yapısı arasındaki iyi yönlü dönüşüm ihtiyacı Go geliştiricileri tarafından sunulan go/parser[1] ve go/printer[2] paketleri ile karşılanmıştır. Dilin resmi Kod-AST dönüştürücülerinin GP'ye uyarlanmasının hem yöntemi benzer dillere uyarlamayı hem de dilin yeni versiyonlarında eklenecek yapıları çözüme eklemeyi kolaylaştıracağı öngörülebilir. Parser ve Printer paketleri arasındaki ilişki Şekil 4.1'de gösterilmiştir. Çizelge 4.1'de ayrıştırıcının Go kodunu AST yapısıyla temsil etmek için kullandığı düğüm tipleri listelenmiştir.

[1] https://pkg.go.dev/go/parser

[2] https://pkg.go.dev/go/printer



Şekil 4.1 Go kodu ve AST arasındaki dönüşümler

Çizelge 4.1 go/ast (v1.20) içinde tanımlanmış olan AST düğüm tipleri

Deyimler	**İfadeler**	**Tip İfadeleri**	**Belirtimler**	**Bildirimler**
AssignStmt	BadExpr	ArrayType	ImportSpec	BadDecl
BadStmt	BasicLit	ChanType	TypeSpec	FuncDecl
BlockStmt	BinaryExpr	FuncType	ValueSpec	GenDecl
BranchStmt	CallExpr	InterfaceType		
CaseClause	CompositeLit	MapType		
CommClause	Ellipsis	StructType		
DeclStmt	FuncLit			
DeferStmt	Ident			
EmptyStmt	IndexExpr			
ExprStmt	IndexListExpr			
ForStmt	KeyValueExpr			
GoStmt	ParenExpr			
IfStmt	SelectorExpr			
IncDecStmt	SliceExpr			
LabeledStmt	StarExpr			
RangeStmt	TypeAssertExpr			
ReturnStmt	UnaryExpr			
SelectStmt				
SendStmt				
SwitchStmt				
TypeSwitchStmt				

ST-ASTGP'nin gereksinimi olan dil tanımındaki tip güvenliği kurallarıyla uyumlu denekler geliştirmeyi mümkün kılmak için go/types[1] paketinin işlevlerini daha basit bir kullanımla sunan golang.org/x/tools/packages[2] paketinden faydalanılmıştır. Bu paketteki packages.Load()[3] fonksiyonu paketler içindeki sembollerin tiplerinin taranması için, types paketindeki types.AssignableTo() fonksiyonu ise AST üzerinde gerçekleştirilecek düzenlemelerin dil tanımındaki tip uyumluluğu kurallarını gözetmek için kullanılmıştır.

1 https://pkg.go.dev/go/types

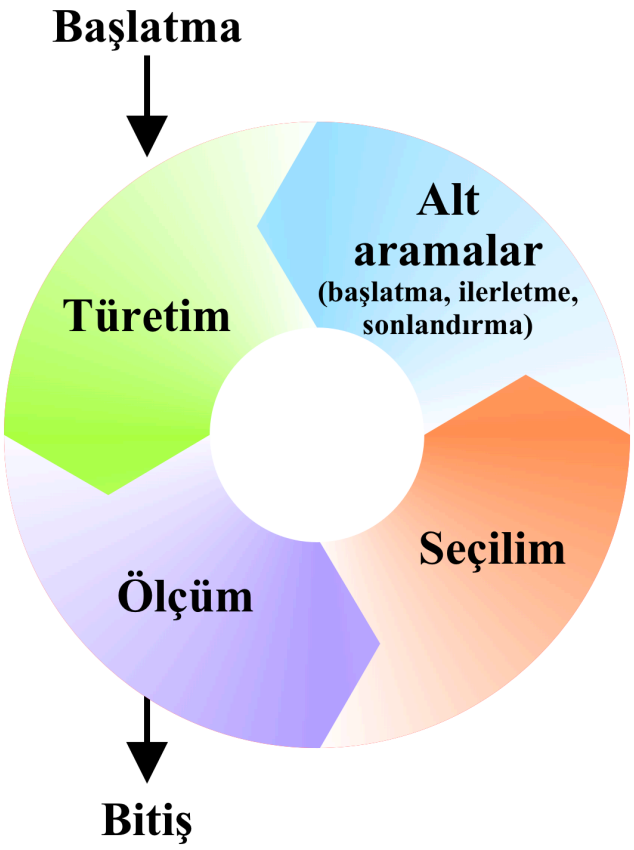
2 https://pkg.go.dev/golang.org/x/tools/go/packages

3 https://cs.opensource.google/go/x/tools/+/v0.16.1:go/packages/packages.go;l=260

5. YÖNTEM

Tanıtılan yöntem Test Güdümlü Evrim (Test Driven Evolution) olarak adlandırılmıştır. TDE, deneklerin temsili için AST'leri kullanır (ASTGP). TDE, tüm ihtimaller arasında en yalın yapılı ve yürütüm, sözdizimi ve baskı hatası bulunmayan gerçeklemeye ulaşma ihtimalini artırmak için sömürüsel karakterli alt aramaları ve keşifsel karakterlerli ana aramayı ayrı ayrı yürüten ve bu çalışmada tanıtılan Katmanlı Arama algoritmasını takip eder. STGP'yi ASTGP'e uyarlayan ST-ASTGP ile arama süresini kısaltmak ve hedefe ulaşma ihtimalini yükseltmek için mutasyonlar sırasında tip güvenliğini korumayı hedefler. Bu sayede geçersiz program, kod ve AST üretimini azaltır ve havuzu hatalı kümelenmelerden korur. ST-ASTGP'nin alt ağaç üretimini gerektiren mutasyonlar sırasında kod seviyesinde tip güvenliğini sağlamaktan sorumlu bileşeni CA-STG dil tanımındaki, standart kütüphanedeki paketlerdeki ve hedef birimin yer aldığı kullanıcı modülündeki paketlerdeki fonksiyon, tip tanımı ve değişkenleri tarayarak mutasyon ile deneğe eklenebilmesine imkan tanır. TDE, Go'nun derlemeli ve zorunlu yapısını destekler.

Özetle, TDE bir birim-testi için çalıştırıldığında test içinde geçen hedef fonksiyonu ve onun bulunduğu dosyayı saptar. Hedefin AST'sinden genetik operasyonlar uygulanarak yeni denekler üretilir. Deneklerin başarım değerlerini belirlemek için basım, derleme, çalıştırma süreci izlenir. Tüm aşamaları geçip testi erken sonlanmadan tamamlayan deneklerin 4 adet başarım değeri oluşur. Seçilim algoritması aynı soydan farklı yönlere evrilen deneklerin arasından bir sonraki aşamaya geçip başarım değerini artıran biri ortaya çıktığında onun ataları hariç diğerlerini eler. Bu sayede işleyişte pozitif fark oluşturması için temsilde birden fazla değişim gerektiren düzenlemelerin oluşumu için imkan tanınır. Ana aramanın döngüsü Şekil 5.1'de gösterilmiştir.



Şekil 5.1 Ana arama adımları

Basılamayan, derlenemeyen veya çalışmasını çalışma zamanı hatası vermeden tamamlayamayan deneklerin bu sorunlarını giderecek genetik düzenlemeleri alma sürecinde başarım değerlerinde görülen pozitif değişimin deneğin kullanıcı tarafından sağlanan problemlerde göstermesi gereken ilerleme ile alakasız olması nedeniyle katmanlı arama yöntemi tasarlanmıştır. Buna göre tamamı basım, derleme ve çalıştırma aşamalarının tümünü geçen denekler arasındaki evrim sürecinde bu aşamalardan herhangi birini geçemeyen bir deneğin oluşması durumunda onun bu şartların tamamını sağlayan en yakınındaki deneği bulmak üzere lokal bir alt arama başlatılır. Alt aramaların ana aramadan ayrıştırılmasıyla ana evrimin aramayı gerektiğinde dar bir alanda yoğunlaştırma (sömürüsel) zorunluluğunun önüne geçilmesi dolayısıyla ana evrime global arama karakteristiği kazandırılması hedeflenmiştir. Bu sayede, sürekli birbirinden farklı yönlere ilerleyen daha ince dallarla çözüm uzayının daha geniş bölgelerinin keşfetmeye meyillendirilmesi amaçlanmıştır (keşifsel).

5.1 Birim Testleri

TDE ile kullanıma uygun bir birim-testi, resmi test aracı ile kullanım için hazırlanan ve örneği Şekil 5.2'te gösterilen birim-testlerinin aksine istenilen ve elde edilen sonuçların karşılaştırmalarında fonksiyon çağrılarını kullanır (Şekil 5.3).

```go
package words

import "testing"

func Test_WordReverse(t *testing.T) {
    cases := map[string]string{
        "yoğurt": "truğoy",
        // ...
    }

    for input, want := range cases {
        got := WordReverse(input)
        if got != want {
            t.Fatalf("WordReverse(%s) = %s (want: %s)", input, got, want)
        }
    }
}
```

Şekil 5.2 Go dilinde standart bir birim-testi

```go
package words

import "tde/pkg/testing"

func TDE_WordReverse(t *testing.T) {
    cases := map[string]string{
        "yoğurt": "truğoy",
        // ...
    }

    for input, want := range cases {
        got := WordReverse(input)
        if !t.Assert(got, want) {
            t.Fatalf("WordReverse(%q) = %q (want: %q)", input, got, want)
        }
    }
}
```

Şekil 5.3 TDE tarafından kullanılabilir formattaki birim-testi

5.2 Semboller Tablosunun Oluşturulması

Semboller; değişkenler, fonksiyonlar, tipler ve operatörlerdir. Mutasyonlar ve çaprazlamalara sırasında başvurulmak üzere pek çok konumda tanımlanmış sembollerin listesi önden oluşturulmalıdır. Bu konumlar:

- Hedef birimin yer aldığı paket,

- Kullanıcı tarafından seçilen; modülde ve GOROOT/src'de bulunan paketler,

- Dil ön tanımlılarını barındıran types.Universe'dir[1].

Sürecin başlangıcında belirtilen hedefler go/types paketi içindeki types.Conf.Check() ile taranır[2], fonksiyon içindeki bir konumdan erişilebilecek tüm kapsamlardaki semboller bir listeye eklenir.

Süreç başlangıcında go/types paketi[3] içindeki fonksiyonu ile hedef birimin yer aldığı paket ve import edilmiş paketlerdeki sembolleri (ast.Ident), bu sembollerin tiplerini (ast.FuncType, ast.MapType vs.) ve tanımlandıkları kapsamları listelemek için kullanılmıştır. Aynı zamanda bu paketin dil yerleşikleri (append, clear vs.) ve ilkel tipler (int, string vs.) için sağladığı sembol ve tip bilgisi kullanılmıştır.

Dizgi tipindeki hata mesajlarını tarayan birim testleri ile sıkça karşılaşılır. Bu string değerlerin karakterlerinin tek tek mutasyonlarla ortaya çıkmasını beklemek yerine birim testlerinden alınmasının verimliliği yükselteceği düşünülmüştür. Dolayısıyla sembol tablosunun oluşturulmasında, birim testteki değerlerden de faydalanılmalıdır.

---

[1] https://cs.opensource.google/go/go/+/refs/tags/go1.21.5:src/go/types/universe.go;l=18;drc=b5515eef565a7d0fd820009fc8c7b282155340a5

[2] https://cs.opensource.google/go/go/+/go1.21.5:src/go/types/api.go;l=423

[3] https://pkg.go.dev/go/types

Çalışmayı katkısıyla orantısız zorlaştıracak özelliklerden kaçılmıştır. İlgi kapsamı dahilinde fmt ve math paketlerinin mutasyonlar tarafından kullanılabilmesine imkan tanınmıştır. Bunun haricinde types.Universe içinde tanımlanmış olan dile dahil operatörler (append, make, new vs.) ve tip tanımları (int, float64, string vs.) de mutasyonlar tarafından tercih edilebilir semboller setine dahil edilir.

5.3 Mutasyonlar

Mutasyonlar 3 düzenlemeden birini gerçekleştirir:

- Rastgele seçilen bir düğümün bir alanına alt ağaç oluşturmak.

- Rastgele seçilen bir düğümün bir alanını ve o alana atanmış alt ağacı silmek.

- Rastgele seçilen bir düğümün düğüm olmayan alanlarındaki değerleri değiştirmek.

Var olan bir alt ağacı düzenlemek, bu işlem sırasında ST-ASTGP'nin gereksinimi olan tip güvenliği ile ilgili gereksinimleri sağlamanın zorluğu sebebiyle tercih edilmemiştir.

Pek çok düğümün birden fazla nitelikleri vardır. Örneğin BinaryExpr düğümü 2 ifadenin sonucunun toplama, çıkarma, çarpma, bölme gibi işlemlerden çıktısını temsil eder. Parametrelere hangi işlemin uygulanacağı düğüm içerisinde jeton tipindeki bir alanda tutulur. Bu değeri değiştirmek iki ifadenin değerleriyle gerçekleştirilecek işlemi değiştirir.

5.4 ST-ASTGP ile Daha İyi Denekler Üretmek

AST'ler ifade ağaçlarının aksine zorunlu dillerin geliştiricinin kullanımına sunduğu zengin özellikleri temsil etme zorunluluğu sebebiyle oldukça karmaşıktır. Bu karmaşıklığın en önemli yan etkisi, gerekli önlemler alınmadığında GP'nin çözüm uzayını aşırı (sunduğu faydanın ötesinde maliyete sebep olacak şekilde) genişletmesidir.

GP'deki hedef karmaşık bir yapıyı deneye yanıla bulmaktır. Evrimin en büyük dezavantajı olan zaman gereksinimi de bu yöntemden kaynaklanır. Bununla birlikte, seçilim algoritmasının yerel minimumlarda takılmayı çözemediği uygulamalarda, yanlış yönde alınan ilerlemelerin tam ve gerçek sonuca ulaşmayı engelleyici hale getireceği düşünülmektedir.

Evrim gibi, zaman maliyeti had safhada olan bir yöntemin genel programcılık sorunları için makül bir çözüm aracı olarak görülmesi ancak var olan her bir zaman kazandırıcı yöntemden yararlanılmasıyla mümkün olabilir. STGP'nin GP'yi hızlandırıcı etkileri tartışılmazdır, dolayısıyla vazgeçilmezdir. STGP'nin ASTGP'ye uygulanmasında AST ve kod üzerindeki tip güvenliğinin farkının tanımlanması gereklidir.

5.4.1 Tip güvenliğini gözetmek

Tip güveliği STGP'nin aksine ST-ASTGP'de 2 yerde aranır: AST ve kod. Kod üzerindeki tip eşleşmesi dil tanımında verilen kurallarla ilgilidir. Örneğin Go'nun statik tipli bir dildir. Dolayısıyla değişkenlere atanan değerler değişkenin tanımlanışındaki tiple uyumlu olmalıdır. Örneğin tanımlanırken sayı tipinde değerler tutabileceği belirtilmiş bir değişkene string atanması kod seviyesinde tip güvenliğinin ihlalidir. AST üzerindeki tip güvenliği ise ağaç içerisindeki her bir düğümün kendi tipini (ifade, deyim, belirtim, bildirim) bekleyen bir düğüme bağlanmasıdır.

Koddaki tip uyumsuzluklarının sonucu kodun programama derlenememesidir; AST üzerindeki tip uyumsuzluklarının sonucu ise AST'nin koda basılamamasıdır. AST üzerindeki tip uyumsuzluklarının sonucuna bir örnek sol el tarafında fonksiyon tanımlamaya çalışılan bir değer ataması deyimi örnek gösterilebilir (Şekil 5.4).

```
&FuncDecl{
    // ...
    Body: &BlockStmt{
        List: []Stmt{
            &AssignStmt{
                // ...
                Lhs: &FuncDecl{}
            }
        }
    }
}
```

Şekil 5.4 İfade tipinde alana bildirim tipinde değer ataması nedeniyle oluşan AST seviyesinde tip uyumsuzluğuna örnek

ST-ASTGP (Strongly-Typed ASTGP) evrimin AST üzerinde gerçekleştirilmesi nedeniyle arama uzayında oluşan genişlemeyi azaltmak için mutasyon ve çaprazlama sırasında hem AST'nin hem de kodun tip kurallarına göre uyumsuz olan düğümlerin bağlanmasını önler.

Yönteme göre, mutasyon ile ekleme yapılacak düğümden önce tanımlanmış tüm değişkenler, fonksiyonlar ve tiplerin listesi çıkartılır. Ardından types.AssignableTo[1] fonksiyonu ile bağlanacakları alanların tip beklentisine uygun olmayan saptanır. Bu seçenekler listeden çıkartılır.

Dille birlikte sunulan statik analiz araçlarının kullanılması ST-ASTGP'nin farklı programlama dillerine uyarlanabilirliğini hayli yükseltir.

[1] https://pkg.go.dev/go/types#AssignableTo

5.5 Çaprazlama

Çaprazlama süreci; ataların çoğaltılması, seçilecek düğümlerin tiplerinin belirlenmesi, belirlenen tipte düğümlerin seçilmesi ve seçilen düğümlerin bağlarının değişiminden ibarettir. Arama uzayındaki genişlemeyi azaltmak amacıyla, çaprazlanacak düğümlerin seçimi sırasında bağlanacakları düğümler arasında AST seviyesinde tip uyumluluğu gözetilmiştir. Çaprazlama iki atanın tüm düğümlerinin tüm alanlarının tiplerinin incelenmesiyle başlar. Hangi tipteki alanlara atanmış düğümlerin seçileceğini belirlemek için iki atada da ortak olarak bulunan alan tipleri sıralanır. Bu liste içinden rastgele bir tip seçilir. İki atadan birer yavru kopyalama ile oluşturulur. İki yavrunun alt ağacında seçilen tipte birer düğüm seçilir. Bu düğümleri işaret eden alanlara diğerinin adresi yazılarak çaprazlama tamamlanır.

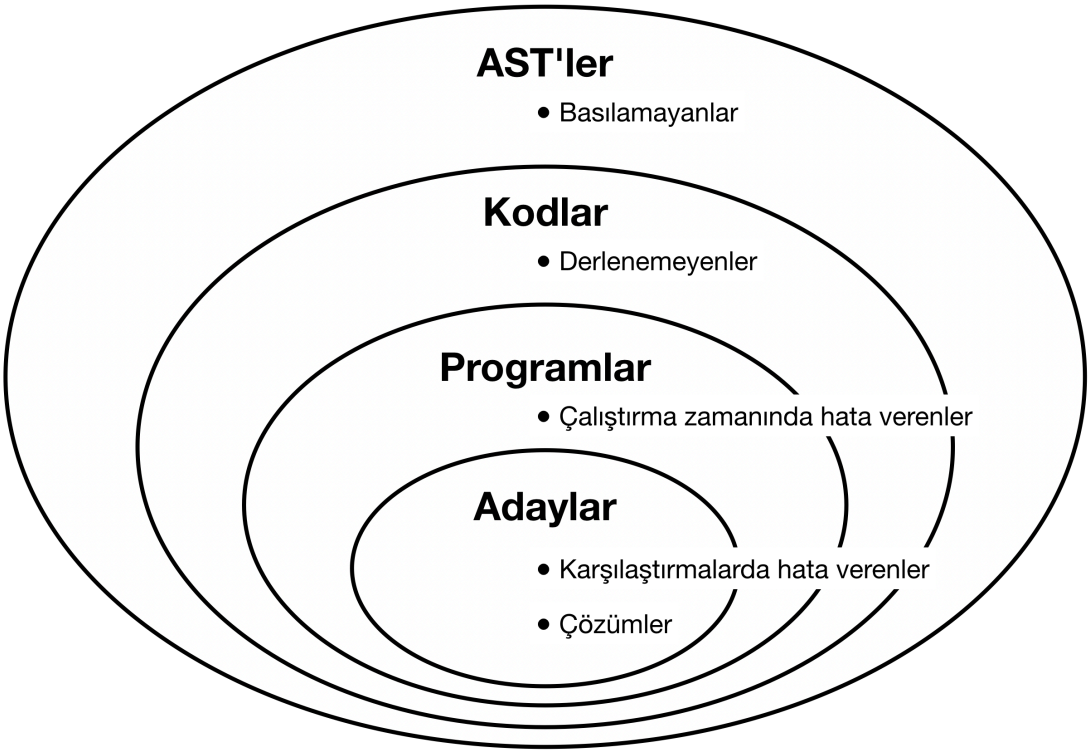
İki denek arasında değiş tokuş edilen alt-ağaçların her birinin tipinin diğer denekte bağlandığı düğümün ilgili alanı tarafından kabul edilen tiplerden biri olması gerekmektedir. Örneğin bir atadaki bir IfStmt'ının koşul alanına bağlanmış BinaryExpr tipindeki bir düğüm diğer atadaki ifade tipindeki alanlardan herhangi birine çaprazlanabilir. Bu kısıt AST seviyesinde tip uyumluluğunu sağlamak için yeterliyken kod seviyesinde tip uyumluluğunu sağlamak için yetersizdir. Örneğin ifade tipinde değerler kabul eden bir başka düğüm tipi olan CallExpr düğümünün işlev alanına BinaryExpr bağlandığında denek basılır ancak derlenemez. Çünkü bu alan özel olarak fonksiyon isimleri veya fonksiyon isimlerine çözülen ifadelerle kullanılmalıdır. Görülmüştür ki AST veri yapılarındaki kısıtlamalar yeterince sıkı belirtilmemiştir. Bu yüzden her AST'nin anlamlı bir kod ifade ettiği söylenemez.

Yazılımsal bir sorunun, bir birim testini geçen birimin birden fazla gerçeklemesi bulunabilir. Örneğin arama gerçekleştiren fonksiyonlar birden fazla algoritma ile gerçeklenebilir. Evrim süreci sırasında bu hedef çözümlerden her birinin soylarının erken nesillerden ayrışan farklı kollardaki aramalarla geliştirileceğini yani bu soylar arasındaki gen alışverişlerinin (çaprazlama vasıtasıyla) çoğu zaman anlamlı olmayacağı

öngörülebilir. Ancak en farklı arama algoritmalarının implementasyonlarında dahi ortak tipte değişken tanımları olduğu görülebilir. Dolayısıyla semantik açıdan hatalı çok uzaktan akraba olan deneklerin, hedeflerin aynılığına bağlı olarak çaprazlanmaları, dolayısıyla aynı havuzda barındırılmaları faydalı olabilir.

5.6 Başarım Değeri

Geçerli her AST geçerli bir kod dosyasına basılamaz. Söz dizimi geçerli olan her kod dosyası derlenemez. Derlenebilen her kod çalıştırılamaz. Çalışan her program çalışmasını tamamlayamaz (Şekil 5.5). Örneğin basım aşamasında yazıcının panik üretmesi kod oluşturulamamasıyla sonuçlanır. Kodun derlenmesi aşamasında derleyici tip eşleşmesi, değişken tanımlamaları vs. kontrolleri sırasında sözdizimi hatası vererek programı oluşturmadan sonlanabilir. Programın (birim testleri vasıtasıyla) çalıştırılması sırasında yetkisiz hafıza erişimi, 0'a bölme vs. nedenlerle testler tamamlanmadan program sonlanabilir. Tüm bu aşamaları geçen deneklerin dahi küçük bir kısmı tüm değer karşılaştırmalarını doğru tamamlayıp çözüm olarak kabul edilebilirler.

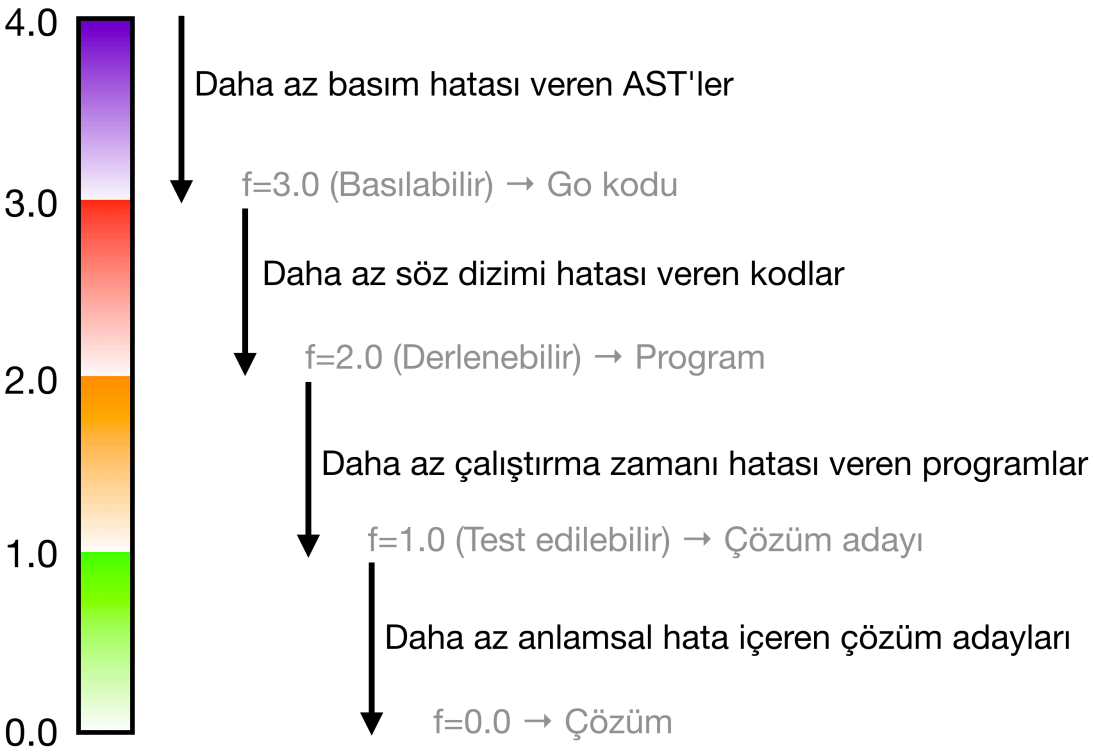


Şekil 5.5 Deneklerin sınıfları arasındaki ilişkileri gösteren venn şeması

Basım, derleme, çalıştırma ve test içindeki hatalar farklı başarım değerine etkiler, birleştirilmezler. Her başarım değeri için 0.0 en iyi sonuçtur. Basım, derleme ve çalıştırma aşamalarındaki başarım değerleri her bir hata için farklı oranlarda artar. Basım hataları resmi printer paketi tarafından döndürülen hatalarla, derleme hataları derleyici tarafından bulunan hatalarla, çalıştırma hataları çalışma zamanı hatalarıyla (runtime error), sınama hataları da örnek durumların sınanmasındaki her bir karşılaştırma ile ilişkilidir.

Bir denek eğer basım aşamasında takılıyor ve Go kodu cinsinden karşılığı oluşmuyorsa başarım değeri 3.0 ila 4.0 arasındadır. Eğer basılabilen bir denek derleme aşamasındayken derleyici tarafından hata alıyorsa başarım değeri 2.0 ile 3.0 arasındadır. Derlenmiş bir denek testlerin yürütümü sırasında çalışma zamanı hatası (örneğin 0'a bölme hatasının sonucu) veriyorsa başarım değeri 1.0 ile 2.0 arasındadır. Bu aşamaların

hepsini tamamlamış bir deneğin başarım değeri ise kullanıcı tarafından birim-testine gömülmüş olan problemlerin ne kadarını doğru hesaplayabildiğine göreceli olarak 0.0 ile 1.0 arasındadır. Bir deneğin alabileceği başarım değerlerinin ne anlama geldiği Şekil 5.6'da gösterilmiştir.



Şekil 5.6 Başarım değerinin anlamlandırılması

5.7 Katmanlı Arama

İki komşu "program denek" arasında çok sayıda "kod denek" ve "AST denek" bulunabilir. Bu durumun sebebi çalışma zamanı hatalarının çözümünün çoğu zaman kod seviyesinde dahi birden fazla satır değişikliği gerektirmesidir (Çizelge 5.1). Çoğu düğüm için AST üzerindeki birden fazla ekleme/çıkarma kodda küçük değişiklikler oluşturur. Üstelik, komşuya ulaşan soyda başarısız olan pek çok yeni soy oluşumuyla karşılaşılacaktır.

Çizelge 5.1 Go dilinde yazılmış programların çalışması sırasında sıklıkla alınan çalıştırma zamanı hataları ve en kısa çözümün uygulanması halinde sorunun çözüleceği durumlar için gerçekleşmesi gerektiği tahmin edilen mutasyon sayısı

Hata	**Çözüm**	**Düğümler**	**Mutasyon**
panic: runtime error: index out of range [X] with length Y	Uzunluk, değer veya Nil kontrolü	IfStmt, BinaryExpr, CallExpr, builtin, BasicLit, ReturnStmt/ BranchStmt	>10
panic: runtime error: invalid memory address or nil pointer dereference			
panic: runtime error: integer divide by zero			
panic: runtime error: slice bounds out of range [:X] with capacity Y			
panic: close of nil channel			
panic: assignment to entry in nil map			
panic: runtime error: makeslice: len out of range	Değer düşürme	BasicLit	1
fatal error: all goroutines are asleep - deadlock!	Kanala değer yazımı ekleme/ çıkarma	UnaryExpr, ChanDir, BasicLit	1, >3
panic: runtime error: concurrent map writes			

İki komşu "aday" arasında değişken sayıda; çözüm adayı olmayan AST, kod ve program denek bulunabilir. Örneğin "undefined: [variable_name]" hatasının giderildiği deneğin bir derinlikli aramanın menzilinde yer alması beklenmez. Çünkü erişilen değişkeni tanımlamak üzere oluşturulacak düğümlerin sayısı biri geçmektedir (bkz. 2.5). Sık karşılaşılan derleme hataları ve en kısa çözümlerin uygulanması için gerekli mutasyon sayısı Çizelge 5.2'de verilmiştir.

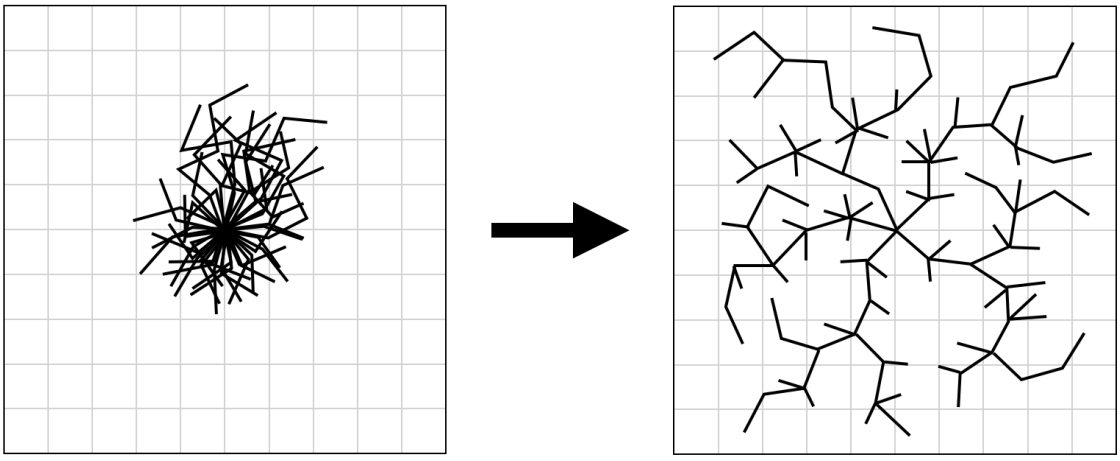
Çizelge 5.2 Go kodlarının derlenmesi sırasında sıklıkla alınan sözdizimi hataları ve en kısa çözümün uygulanması halinde sorunun çözüleceği durumlar için gerçekleşmesi gerektiği tahmin edilen mutasyon sayısı

<table>
<tr><th>Hata</th><th>Çözüm</th><th>Düğümler</th><th>Mutasyon</th></tr>
<tr><td>undefined: [variable_name]</td><td>Değişken tanımlama</td><td>AssignStmt/ DeclStmt</td><td>>4</td></tr>
<tr><td>invalid operation: mismatched types int and string</td><td>Tanım düzenleme</td><td>DeclStmt, Ident</td><td>1</td></tr>
<tr><td>missing return at end of function</td><td colspan="3">ReturnStmt kullanıcıdan beklendiği için kapsam dışındaki hata</td></tr>
<tr><td>cannot find package "[package_name]" in any of:</td><td colspan="3" rowspan="6">AST'den go/printer ile basılmış kodda oluşmayacağı için kapsam dışında kalan hatalar</td></tr>
<tr><td>expected 'package', found 'EOF'</td></tr>
<tr><td>undefined: [PackageName]</td></tr>
<tr><td>imported and not used: "[PackageName]"</td></tr>
<tr><td>syntax error: unexpected [token], expecting [token]</td></tr>
<tr><td>syntax error: unexpected name, expecting semicolon or newline</td></tr>
</table>

Aynı şekilde iki komşu "kod denek" arasında da nadiren de olsa birden fazla AST denek bulunabilir.

Özetle, semantik hataların giderimi sürecinde önceki sınama adımlarında takılan deneklerin üzerinden geçilmemesi göz ardı edilebilecek kadar düşük bir ihtimaldir. Dolayısıyla zorunlu ve derlemeli yapıdaki bir dil için çalışması beklenen bir GP bu tür n adımlı iyileştirmeleri yani kompleks adaptasyonları geliştirme ihtimali olan soyları prematüre elemeye karşı korumak zorundadır.

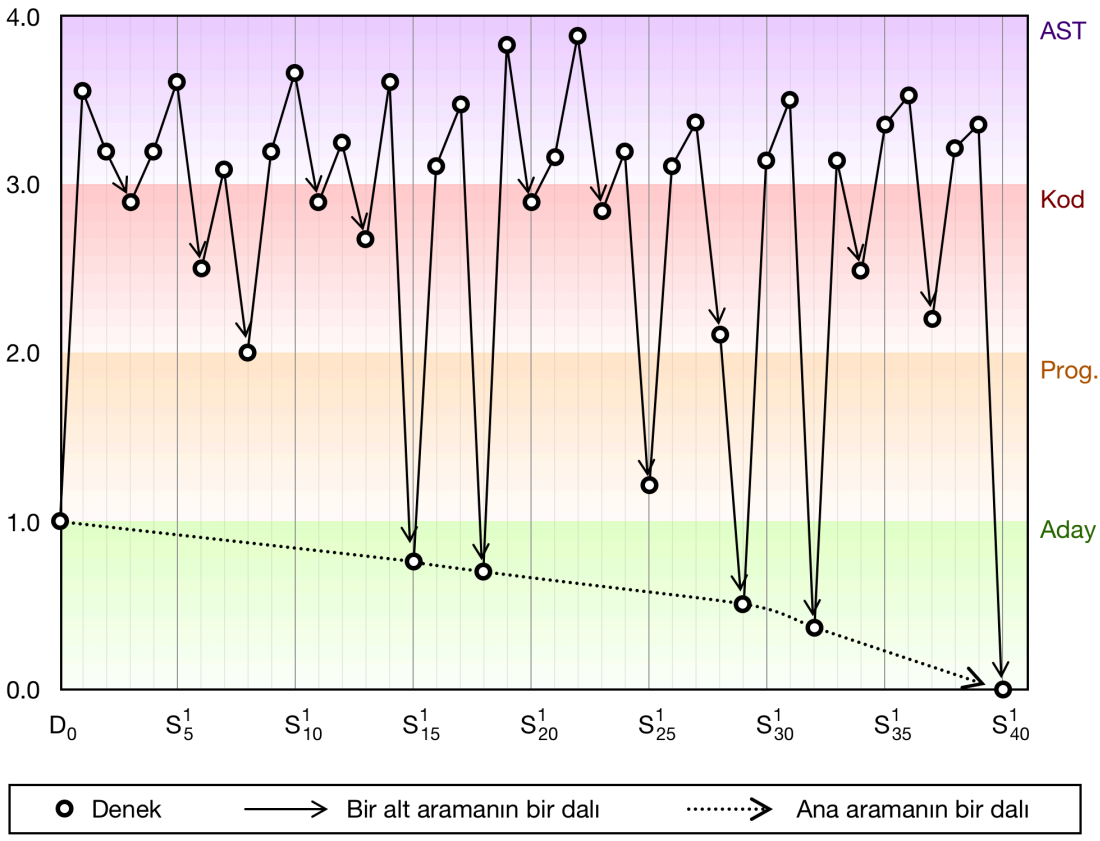
Katmanlı Arama algoritması, özünde, ölçme aşamaları arasındaki bir önceki aşamada takılan bir yavru ortaya çıktığında, sadece ilgili seviyedeki hataları azaltmaya hedeflenmiş alt evrimler (katmanlar) başlatılır. Bu alt evrimlerin nesil, derinlik ve toplam ölçüm kotaları ana evrimden daha sınırlıdır.



Şekil 5.7 Katmanlı aramanın arama karakterinde oluşturmayı hedeflediği değişim

Sürece göre, başlangıçta kullanıcı tarafından fonksiyon adı, girdi/çıktı parametre listesi ve varsayılan değerlerle oluşturulmuş Return deyiminden ibaret olarak birim alınır. Bu birime başlangıç noktası (D_0) denir.

Tanıtılan yöntemin çalışması boyunca onları elemek yerine üzerinden evrimleştirmeye devam etmesi gereken daha düşük başarımlı deneklerin çözümle ilişkisini temsilen Şekil 5.8'deki grafik çizilmiştir. Bu şekilde S_{29}'dan S_{64}'e ilerlenen evrimsel süreçte 30 denek üretildiği, bunlardan sadece birinin aday statüsüne eriştiği, başka birinin program statüsüne eriştiği ve bunların dışında sadece ikisinin kod statüsüne eriştiği görülebilir. Diğer 26 denek ise koda dönüştürülemeyen geçersiz AST'ler olmuşlardır.



Şekil 5.8 Yöntemin sonuca daha başarısız ara formlar üzerinden ulaşabilmesi gereken kod sentezi sorununda Katmanlı Arama'nın işleyişinin sonuca (S^1_{40}) ulaşılmış bir soy (S) boyunca uğradığı ara formlar üzerinden gösterimi

Bu gösterimde dikkat çekmesi gereken şey çözüm adaylarını doğrudan birbirine bağlayan çizginin başlangıçtan bitişe doğru meyillidir. Çözüme rastgele nesillerdeki ilerlemelerle yaklaşılmış ancak hiçbir zaman uzaklaşılmamıştır (stokastik ilerleme). Bu

durum Katmanlı Aramanın çözüm adayı olmayan denekleri ana havuzdan izole havuzlarda tutmasının nedenidir. Baskı, sözdizimi ve yürütüm sorunlarının giderimi için gerekli değişiklikler lokal aramaların konusudur. Çünkü bir çözüm adayından türeyen başarısız yavrunun sözdizimi hatalarını gidermek için gerekli değişiklikler bir başka çözüm adayınınkinden farklıdır. Bu hataları gidermek için yürütülen evrimsel araştırmanın sonucu bilgi, genel kullanıma sahip değildir.

Tasarlanan yöntem Katmanlı Arama, kompleks adaptasyonların ortaya çıkışını mümkün kılmayı hedeflerken ana aramanın keşifsel yapısını koruma çalışır. Bu hedefe çalışma zamanı, sözdizimi ve baskı hatalarının çözümünü, yerel olarak, daha sığ ve sık dallanmalı, hızlı ve izole havuza sahip alt evrimler içinde arayarak ulaşmaya çalışır.

Ana arama bir döngüdür ve adımları türetim, ölçme, seçilim ile alt aramaların başlatılması, ilerletilmesi ve sonlandırılmasıdır. Ana aramanın havuzunda sadece çözüm adayları bulunur. Türetim evresinde havuzdaki çözüm adaylarından başarımlarına orantılı ihtimallerle seçilen deneklerden alınan klonlar mutasyona ve çaprazlamalara tabi tutularak yavruları oluşturur. Ölçme evresinde, neslin her yeni yavrusu sınama sürecindeki adımlardan geçerek başarımı ölçülür. Sonrasında f=0.0 olan deneğin varlığı kontrol edilir. Bu çözüm bulunduğunda arama sonlanır ve çözümü temsil eden AST döndürülür. Diğer durumda seçilim süreci başlar. Parametrelere göre hesaplanan sayıda aday Roulette Wheel algoritması ile seçilerek havuzdan silinir. Yavrulardan hayatta kalan AST, kod, program denekler için alt-aramalar başlatılır. Önceden gelen alt-aramalar bir nesil ilerletilir. Parametrelerce belirtilen nesil limitlerine ulaşan ancak ürün verememiş alt aramalar erken sonlandırılır ve havuzlarındaki deneklerle birlikte silinir.

Tanımlanan alt aramalar 4 çeşittir. İsimleri:

- Sadece çözüm adayı statüsündeki deneklerle çalışan ve çözüm arayan Ana Arama (Şekil 5.9),

- Program denekler üzerinden Ana Arama için çözüm adayları ve üstü arayan alt arama Aday Araması (Şekil 5.10),

- Kod denekler üzerinden Aday Araması için program denekler ve üstü arayan alt arama Program Araması (Şekil 5.11),

- AST denekler üzerinden Program Araması için kod denekler ve üstü arayan alt arama Kod Aramasıdır (Şekil 5.12).

Ana Arama, Aday Araması ve Program Aramaları her yinelemede önceki nesillerde oluşturulmuş alt aramalarını kota dahilinde yineler. Aday, Program ve Kod aramalarının adımları Ana Aramaya benzerdir.

Tüm dallar arasında dengeli ilerlemek için her bir aramanın bir nesilde oluşturabileceği (ve dolayısıyla sınamaya sokabileceği) denek sayısı (L) sınırlandırılmıştır. Eleme fonksiyonu havuzda en fazla ya bir sonraki tur sınanabilecek denek kadar azaltır, ya da nüfus limiti P'ye kadar. Ana arama N. nesle ulaştığında sonlanır. Alt aramalar üst aramaları tarafından kendilerine izin verilen N'lere geldiklerinde sonlandırılırlar.

Girdiler: Kök denek **R**,
Birim-testi U,
Ana arama nesil limiti **N**,
Ana arama nüfus limiti **P**,
Ana arama nesil başına sınama limiti **L,**
Aday araması parametreleri A_N, A_P, A_L,
Program araması parametreleri P_N, P_P, P_L,
Kod araması parametreleri K_N, K_P, K_L
Çıktılar: Çözümlerin AST'leri

```
1.  S çözüm adaylarının listesi olsun.
2.  S ← { R }
3.  AA aday aramalarının listesi olsun.
4.  AA ← { }
5.  for n ← 0 to N do
6.      O ← YavruÜretim(S, min(L, ⌈ |S| / 2 ⌉))
7.      Sınama(U, O)
8.      Çözümler ← { o ∈ O | o.Fitness = 0.0 }
9.      if | Çözümler | > 0 then
10.         return Çözümler
11.     end if
12.     Eleme(S, P, L)
13.     for aa ∈ AA do
14.         çözümler, adaylar ←aa.İlerle(U, A_P, A_L, P_N, P_P, P_L, K_N, K_P, K_L)
15.         if çözümler ≠ ∅ then
16.             return çözümler
17.         else if adaylar ≠ ∅ then
18.             S ← S ∪ adaylar
19.             AA ← AA / aa
20.         else if aa.Nesil == A_N then
21.             AA ← AA / aa
22.         end if
23.     done
24.     Programlar ← { o ∈ O | o.Fitness >= 1.0 }
25.     if | Programlar | > 0 then
26.         AA ← AA ∪ { AdayAraması(o) }
27.     end if
28. done
29. return NIL
```

Şekil 5.9 Katmanlı Arama algoritması

Girdiler: Aday Araması **aa,**
Birim-testi U,
Aday araması nüfus limiti **P**,
Aday araması nesil başına sınama limiti **L,**
Program araması parametreleri P_N, P_P, P_L,
Kod araması parametreleri K_N, K_P, K_L
Çıktılar: Çözümlerin AST'leri, Adayların AST'leri

```
1.  O ← YavruÜretim(aa.S, min(L, ⌈ |aa.S| / 2 ⌉))
2.  Sınama(U, O)
3.  Çözümler ← { o ∈ O | o.Fitness = 0.0 }
4.  Adaylar ← { o ∈ O | 0.0 < o.Fitness <= 1.0 }
5.  if | Çözümler | > 0 ∨ | Adaylar | > 0 then
6.      return Çözümler, Adaylar
7.  end if
8.  Eleme(aa.S, P, L)
9.  for pa ∈ PA do
10.     çözümler, adaylar, programlar ← pa.İlerle(U, P_P, P_L, K_N, K_P, K_L)
11.     if | çözümler | > 0 ∨ | adaylar | > 0 then
12.         return çözümler, adaylar
13.     else if | programlar | > 0 then
14.         aa.S ← aa.S ∪ programlar
15.         aa.PA ← aa.PA / pa
16.     else if pa.Nesil == P_N then
17.         aa.PA ← aa.PA / pa
18.     end if
19. done
20. Kodlar ← { o ∈ O | o.Fitness >= 2.0 }
21. if | Kodlar | > 0 then
22.     aa.PA ← aa.PA ∪ { ProgramAraması(o) }
23. end if
24. aa.Nesil ← aa.Nesil + 1
25. return NIL, NIL
```

Şekil 5.10 AdayAraması.İlerle algoritması

Girdiler: Program Araması **pa,**
Birim-testi U,
Program Araması nüfus limiti **P**,
Program Araması nesil başına sınama limiti **L,**
Kod araması parametreleri K_N, K_P, K_L
Çıktılar: Çözümlerin AST'leri, Adayların AST'leri, Programların AST'leri

```
1.  O ← YavruÜretim(pa.S, min(L, ⌈ |pa.S| / 2 ⌉))
2.  Sınama(U, O)
3.  Çözümler ← { o ∈ O | o.Fitness = 0.0 }
4.  Adaylar ← { o ∈ O | 0.0 < o.Fitness <= 1.0 }
5.  Programlar ← { o ∈ O | 1.0 < o.Fitness <= 2.0 }
6.  if | Çözümler | > 0 ∨ | Adaylar | > 0 ∨ | Programlar | > 0 then
7.      return Çözümler, Adaylar, Programlar
8.  end if
9.  Eleme(pa.S, P, L)
10. for ka ∈ KA do
11.     çözümler, adaylar, programlar, kodlar ← ka.İlerle(U, K_P, K_L)
12.     if | çözümler | > 0 ∨ | adaylar | > 0 ∨ | programlar | > 0 then
13.         return çözümler, adaylar, programlar
14.     else if | kodlar | > 0 then
15.         pa.S ← pa.S ∪ kodlar
16.         pa.KA ← pa.KA / ka
17.     else if ka.Nesil == K_N then
18.         pa.KA ← pa.KA / ka
19.     end if
20. done
21. ASTler ← { o ∈ O | o.Fitness >= 3.0 }
22. if | ASTler | > 0 then
23.     pa.KA ← pa.KA ∪ { KodAraması(o) }
24. end if
25. pa.Nesil ← pa.Nesil + 1
26. return NIL, NIL, NIL
```

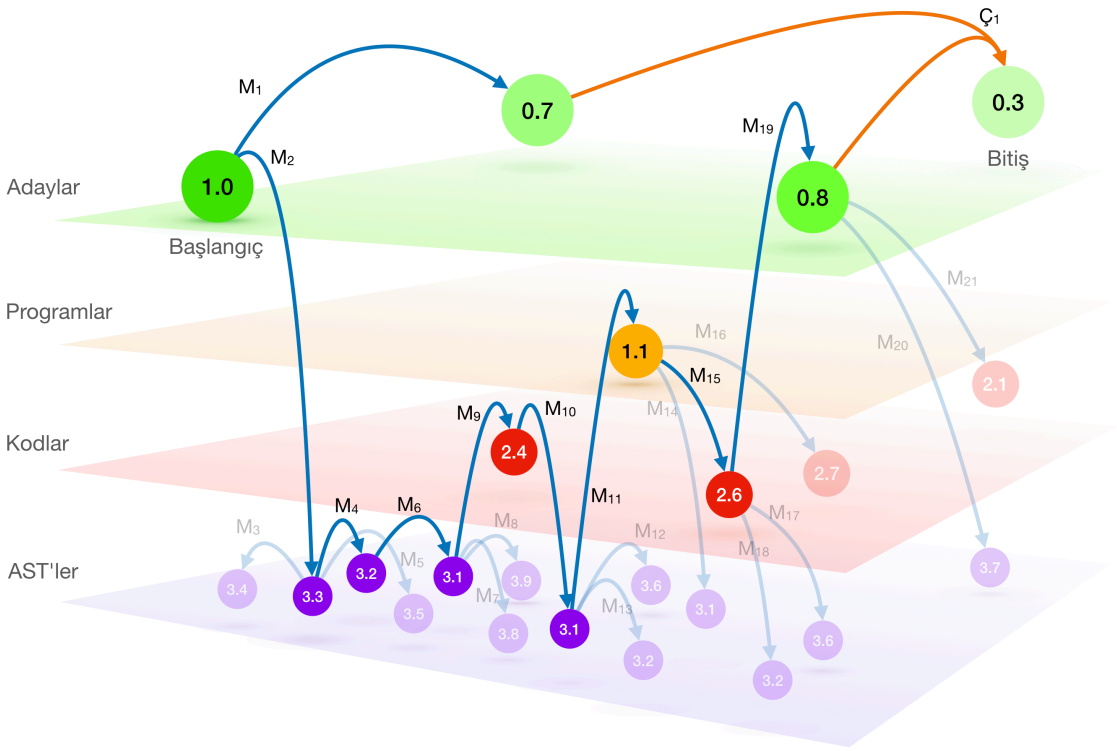
Şekil 5.11 ProgramAraması.İlerle algoritması

Girdiler: Kod Araması **ka,**
Birim-testi *U*,
Kod Araması nüfus limiti **P**,
Kod Araması nesil başına sınama limiti **L**
Çıktılar: Çözümlerin AST'leri,
Adayların AST'leri,
Programların AST'leri
Kodların AST'leri

```
1.  O ← YavruÜretim(ka.S, min(L, ⌈|ka.S|/2⌉))
2.  Sınama(U, O)
3.  Çözümler ← { o ∈ O | o.Fitness = 0.0 }
4.  Adaylar ← { o ∈ O | 0.0 < o.Fitness <= 1.0 }
5.  Programlar ← { o ∈ O | 1.0 < o.Fitness <= 2.0 }
6.  Kodlar ← { o ∈ O | 2.0 < o.Fitness <= 3.0 }
7.  if |Çözümler| > 0 ∨ |Adaylar| > 0 ∨ |Programlar| > 0 ∨ |Kodlar| > 0 then
8.      return Çözümler, Adaylar, Programlar, Kodlar
9.  end if
10. Eleme(ka.S, P, L)
11. ka.Nesil ← ka.Nesil + 1
12. return NIL, NIL, NIL, NIL
```

Şekil 5.12 KodAraması.İlerle algoritması

Şekil 5.13, Şekil 5.8'deki başarıya ulaşan deneğin soyuna odaklanan gösterimin aksine, süreçte elenen deneklerin başarıya ulaşan deneklerle ilişkisini ve katmanlı aramanın ana aramanın havuzunu çözüm adayı olmayan deneklerden arındırdığını gösterir. Mutasyonlar M_i, çaprazlama $Ç_i$ ile gösterilmiştir. Üretilmiş deneklerin çoğunun alt katmanlarda olduğu görülebilir.



Şekil 5.13 Katmanlı aramanın oluşturduğu dallanmalar ve alt aramalar ile ana aramanın ilişkisi

5.8 Deneklerin Programlara Dönüştürülmesi ve Birim Testinin Çalıştırılması

AST formunda saklanan, genetik operasyonlara maruz bırakılan deneklerin başarımlarının ölçülebilmesi için programlara dönüştürülmesi gereklidir. Bu süreç AST'nin koda basımı; basılan kodun birim testi ile birlikte geçici bir klasöre kopyalanması; çalıştırıldığında birim testini çağıracak, test bittiğinde sonuçları diske yazacak testçi kodun kopya klasöre yerleştirilmesi; kodun derlenmesi; programın başlatılması ve sonuçların diskten okunması adımlarından oluşur. Şekil 5.14 süreç boyunca elde edilen yapıları ve adımları göstermektedir.

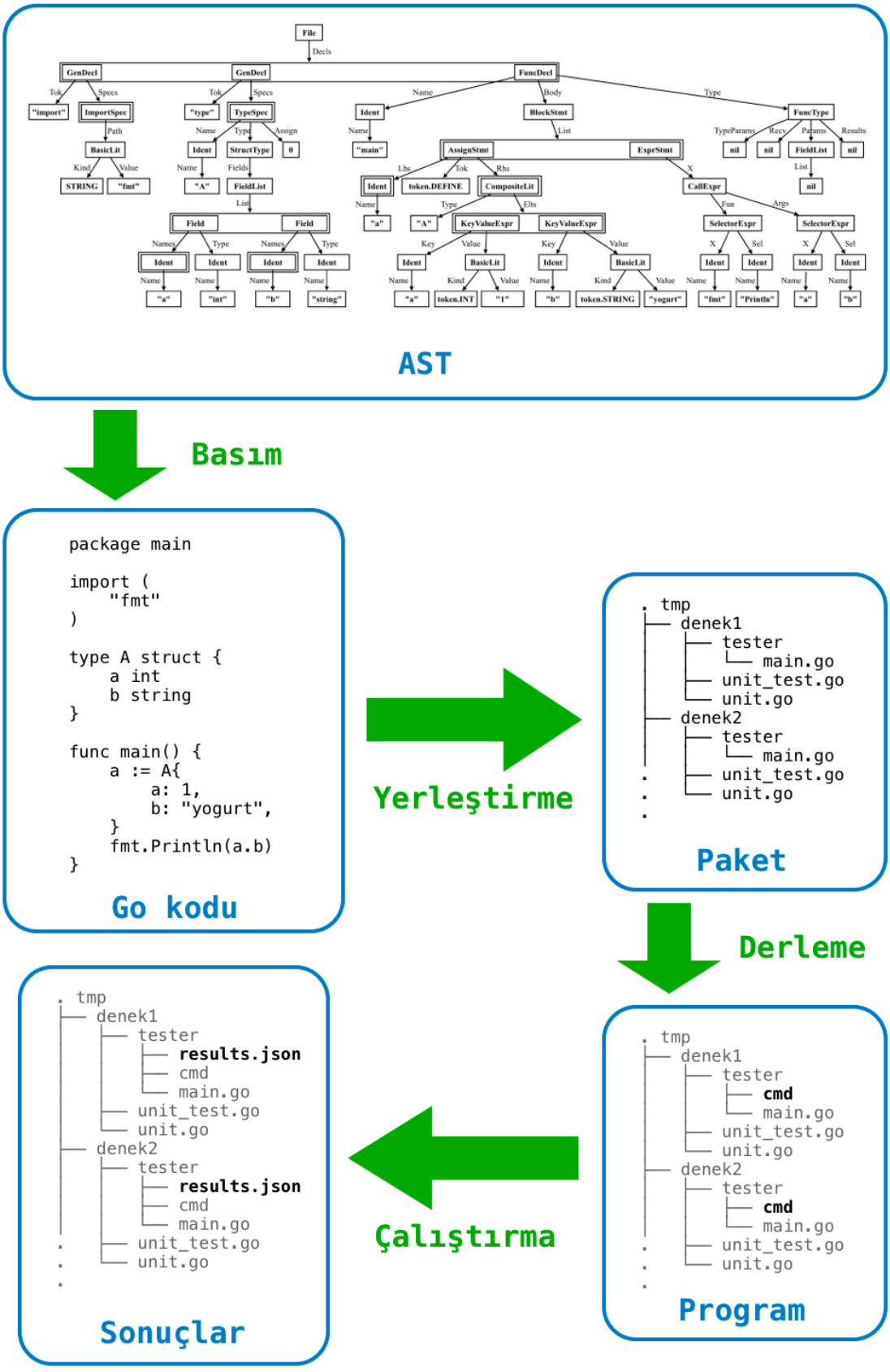
5.8.1 Basım

Bu aşamada her bir AST, bulunduğu dosyanın AST'si içinde resmi printer paketi içerisindeki Fprint fonksiyonu kullanılarak Go koduna çevrilmeye çalışılır. Başarılı sonuç alan deneklerin basım başarımı 0 olarak kaydedilir. Aksi durumda hata sayısına göre (0.0, 1.0] arasında bir başarımı hesaplanır.

5.8.2 Yerleştirme

Denekler modül içerisindeki yapılara erişme ihtimaline sahip oldukları için derleme ve çalıştırma ortamına sadece paket değil bütün paket kopyalanır. Ölçümlenme sırası gelen denek bulunması modülün nüshası içinde gereken dosyaya yazılır.

Birim-testler doğrudan yürütülebilir dosyalara derlenemezler, dolayısıyla doğrudan çalıştırılamazlar. Bu yüzden paket içinde birim-testi çağıracak bir alt-paket oluşturulur (tester).



Şekil 5.14 Ölçümleme süreci adımları

5.8.3 Derleme

Tester klasöründeki paket main olarak isimlendirilmiştir, dolayısıyla standart derleyici yardımıyla bir yürütülebilire derlenebilir.

5.8.4 Çalıştırma

Her bir deneğin AST'si hedefin bulunduğu orijinal paketin TDE tarafından kontrol edilen dizinde oluşturulan bir nüshasına yazılır. Bu nüshaların her birinde birim-testi çalıştıracak bir alt-paket oluşturulur. Bu alt-paket farklı olarak main adını taşır ve resmi Go derleyicisi tarafından çalıştırılabilir bir dosyaya derlenir.

Şekil 5.3'te görüldüğü üzere birim-testler bu tür, testle ilgili detayların birim-test tarafından işleneceği bir parametre alır. Go dili ve standart testing aracı için yazılan testler için bu parametrenin tipi testing.T'dir. Testçi paket derlenip çalıştırıldığında özel bir "T" örneği oluşturur. Parametre olarak kullanarak birim testi çağırır. Birim test her karşılaştırmada elde edilen ve beklenen değerleri girdi parametresi olarak kullanarak t.Assert metodunu çağırır. Assert metodu iki değer arasındaki uzaklığı hesaplayarak T örneği içinde tutar. Birim test döndükten sonra testçi paket genel hesaplamayı yapar.

5.9 Karşılaştırmaların Gerçekleştirilmesi ve Aday Başarısının Ölçülmesi

Test Güdümlü Geliştirme amacıyla yazılan birim testlerinde bulunan değer karşılaştırmaları, istenen ve döndürülen değerler arasındaki aynılık/farklılık (Boole) sınıflandırmasını gerçekleştirir. Fakat bu karşılaştırma sonuçları birim testlerini başarım fonksiyonu olarak kullanmak için yeterli değildir. Çünkü bir başarım fonksiyonu, deneğin üretmesi gereken sonuçlara ne kadar yaklaştığını ölçer. Bu sayede başarım fonksiyonu iki denekten daha kötü olanı elemede ölçüt olarak kullanılabilir hale gelir.

Seçilim evresine geçebilmek için testlerin tamamlanmasının ardından her bir deneğin birim testinde bulunan örnek girdiler için ürettiği değerlerin istenilen değerlere yakınlığı ölçülmeli ve ardından tüm denekler arasında daha "iyi" olan denekler karşılaştırma yoluyla belirlenmelidir.

Üretilen değer yakınlığını integer ve floating-point tipindeki değerler için ölçmek oldukça kolaydır. Basit bir $|a - b|$ işlemi ile yakınlık hesabı gerçekleştirilir. Öte yandan diğer tipteki değerler için yol bu kadar bariz değildir. Örneğin dizginin için farklı yöntemler bulunmaktadır. Bunlardan biri soldan sağa doğru iki dizginin aynı indisteki karakterlerini karşılaştırmaktır. Bu yöntemin farklı uzunluktaki değerler için gerçekleştirmesi gereken ideal davranış belirsizdir. Başka bir yöntem Levenshtein uzaklığı algoritmasını kullanmaktır. Bu yöntem (n ve m, iki dizginin uzunluğu olmak üzere) $O(nm)$ zaman karmaşıklığına sahiptir. Çoğu birim testinin karşılaştırdığı değerlerin kodda yer aldığı ve okunabilirlik kaygısıyla kısa tutulduğu düşünüldüğünde sorun olarak görülmez. Dosya içeriği karşılaştırmaları için tercih edilmesi önerilmez. Kullanıcıdan tanımlanan özel tiplerdeki değerlerin yani bileşik değerlerin karşılaştırması bu tiplerin ilkel tipteki alanlarının karşılaştırması üzerinden ayrı ayrı yapılır (bkz. 8.5). Slice ve map değer karşılaştırmaları için aynı indisteki ve anahtardaki değerlerin varlığı ve yakınlığı gözetilir.

5.10 Sınama Sonuçlarının Toplanması

Birim-testi çalıştıran program değer karşılaştırması sonuçlarını deneğin kimlik numarasıyla birlikte diske yazar. Test sonlandıktan sonra yürütme evrimi yöneten programa geçer. Tüm deneklerin test sonuçları dosyadan okunduğunda eleme aşamasına geçilir.

5.11 Eleme

Nüfusu sabit tutmak için eleme uygulanır. Çeşitliliği tamamen kaybetmeden nesilden nesle gerilememeyi sağlamak hedefiyle başarım orantılı seçilim (Fitness Proportionate Selection) algoritması olan Rulet Çarkı ve her nesildeki en iyi denekleri koşulsuz korumayı sağlayan Onur Listesi[1] yöntemi uygulanmıştır.

[1] https://deap.readthedocs.io/en/master/api/tools.html#hall-of-fame

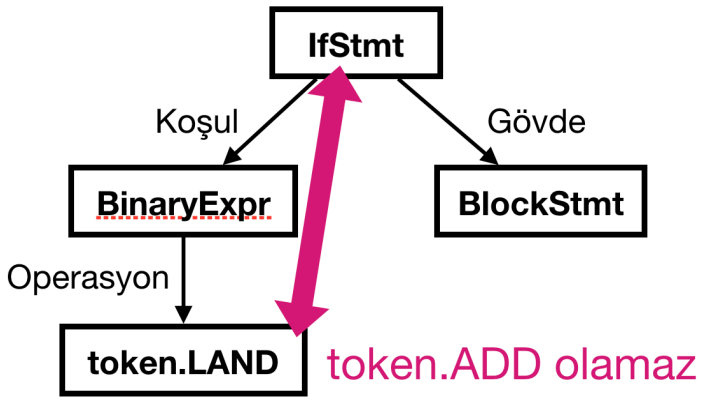
6. BULGULAR

6.1 ASTGP

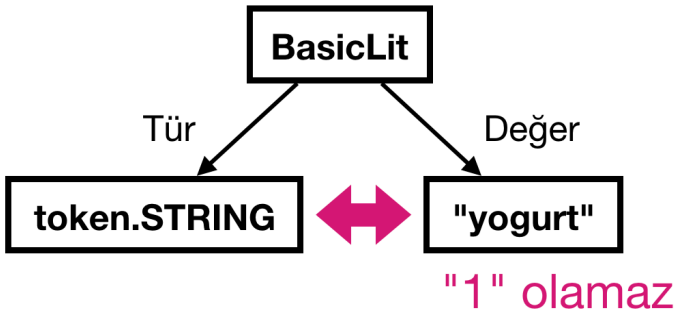
Programları temsil etmek için Koza tarzı ifade ağaçları yerine AST'nin kullanımının neden çözüm uzayının aşırı büyümesine sebep olduğu görülmüştür. Sebepler: evrim sırasında koda dönüştürülemeyen geçersiz AST'lerin ortaya çıkması, AST'nin fonksiyonun girdileri ile çıktıları arasındaki ilişki yerine kod akışını temsil etmesi nedeniyle oluşan dolaylılık ve neticesinde artan alternatif temsil sayısı, dillerin zengin özellik seti nedeniyle her mutasyonda içinden doğru operatörü/terminali seçmek için daha fazla aday olmasıdır.

6.2 STGP

Görülmüştür ki ast-üst arasında tip uyuşmasını gözeterek uyumsuz deneklerin oluşumunu engelleyen ve çözüm uzayının daraltılmasını sağlayan STGP'nin AST ile gerçeklenen GP'ye doğrudan uygulanması yeterli daraltmayı getirmemiştir. Bunun bir sebebi ASTGP'deki asıl uzay daraltmasını sağlayacak tip uyum gözetiminin doğrudan birinci derece ast-üst arasında değil AST'nin herhangi bir alt ağacı içerisindeki düğüm-ata (Şekil 6.1) veya aynı ataya bağlı düğümler arasında (Şekil 6.2) gözetilmesi gerekliliği saptanmıştır.



Şekil 6.1 IfStmt türündeki düğümün 2. dereden astının alabileceği değerleri sınırlaması



Şekil 6.2 BasicLit türündeki bir düğümün bir astının değerinin diğer astınının alabileceği değerleri sınırlaması

6.3 ST-ASTGP

Dil kurallarının gerektirdiği, kod seviyesinde tip uyumunu sağlamayan STG'nin ürettiği mutasyonların doğru yapıları oluşturmak için çok daha derin evrimsel aramalar gerçekleştirmesi gerektiği görülmüştür (EK 2). Dolayısıyla ST-ASTGP'nin yoksunluğunda aramanın daha çok keşifsel meyilli yapılandırılması gerekeceği görülmüştür. Tabi ki bu durumda hiçbir yerel/global minimuma erişememe ihtimali yükselecektir. Özet olarak görülmüştür ki zorunlu ve derlemeli dillerde GP'nin

uygulanabilmesi için hem daha iyi denekler üretmek üzere yöntemlerin geliştirilmesi önemlidir.

6.4 Derleme Süresi

Derleme işlemi tüm sürecin en çok zaman alan kısmıdır. Özellikle disk erişimi nedeniyle. Derleyicinin süreci yöneten programa entegre edilmesi ve denekleri diske yazmadan derleyebilme imkanı süreci (katsayısal anlamda) kısaltma potansiyeline sahiptir.

Paralelleştirme ve disk işlemlerinin toplam süreye etkisini görmek için bir test düzenlenmiştir. Bu testte bir Go programının derlenmesini sağlayan komut, /dev/null hedefine derleyen komut, tmp dizinine derleyip oradan çalıştırılmasını sağlayan komut ve derlediği binary'i PATH'e dahil bir dizine ekleyen komut art arda ve paralel olarak basit bir program (Merhaba Dünya) için 100 ve oldukça gelişmiş olan açık kaynaklı bir proje (terraform, commit 8402b97) için 100 örnek çalıştırılmıştır. Zaman ölçümleri için UNIX/time aracı kullanılmıştır. Bu araç tamamlanma süresini ölçtüğü komut için 3 değer sunmaktadır: System, User, Real. System işlemcinin çekirdekte gerçekleşen işlemler sırasında geçirdiği süre, User işlemcinin kullanıcı kipindeki işlemler sırasında geçirdiği süre, Real ise duvar saati süresidir (Anonymous 2019). System ve User toplamının Real ile farklı olmasının sebebi işlemci çekirdeği sayısı ve işlemciden haricindeki faaliyetlerdir (örneğin disk erişimi).

Çizelge 6.1 ve 6.2'deki ölçümler karşılaştırıldığında pek çok fikir oluşmaktadır. Go derleyicisinin paralelleştirilmesi kullanıcı kipinde gerçekleşen CPU faaliyetini artırmaktadır. Büyük programların derlenmesinde bu faaliyet artışı çekirdek avantajına rağmen toplam süreyi artırmaktadır.

Hafızanın disk gibi kullanılmasına imkan sağlayan bir araç ile 100 seri Merhaba Dünya programı derleme denemesi tekrarlanmıştır. Bu araç sayesinde hafıza hem derleyiciye

verilen kaynak kodlarını tutmak için hem de derleyici tarafından derleme sırasında oluşturulan ara-ürünlerin saklanması için kullanılmıştır. Derleyicinin aynı şekilde yürütülebilir dosyayı hafızadaki konuma üretmesi istenmiştir. Sonuç olarak bu denemede de alınan sürenin (8,7s), standart ayarlarla yapılan derlemeye (10,7s) kıyasla 19% kısalma sağladığı görülmüştür. Dolayısıyla disk operasyonlarının derleme (dolayısıyla ölçümleme) sürecine etkisi azaltılmıştır.

Çizelge 6.1 Merhaba Dünya programı için derleme başına alınan gerçek ve işlemci süreleri (100 örnek)

komut	real (sn)	user (sn)	sys (sn)	user+sys (sn)
go build -o /dev/null .	0,19	0,25	0,29	0,54
go build -o /dev/null . &	0,05	0,36	0,33	0,69
go build .	0,1	0,15	0,26	0,41
go build . &	0,05	0,37	0,32	0,69
go run . >/dev/null 2>&1	0,25	0,2	0,28	0,48
go run . >/dev/null 2>&1 &	0,12	0,3	0,3	0,6
go install -v .	0,13	0,18	0,31	0,49
go install -v . &	0,05	0,38	0,36	0,74

Çizelge 6.2 Terraform programı için derleme başına alınan gerçek ve işlemci süreleri (10 örnek)

komut	real (sn)	user (sn)	sys (sn)	user+sys (sn)
go build -o /dev/null .	4,3	7,2	2,8	10
go build -o /dev/null . &	7,9	78,1	17	95,1
go build .	0,6	2,2	1,9	4,1
go build . &	1	9,3	5,4	14,7
go run . >/dev/null 2>&1	3,8	4,7	2,2	6,9
go run . >/dev/null 2>&1 &	2,3	8,5	7,1	15,6
go install -v .	0,6	2,2	1,9	4,1
go install -v . &	1	9,5	5,3	14,8

7. TARTIŞMA

7.1 Evrimsel Algoritmalar ve Biyolojik Evrim Karşılaştırması

Evrimsel algoritmalar (GA, GP, NE vs.) biyolojik evrimin karmaşık mekanizmalarından farklı olarak mutasyonlar sırasında tamamen rastgele değişiklikler üretir. Ancak hem önceki çalışmalarda hem de bu tezin konusu olan deneylerde görülmüştür ki daha uygun denekleri üretmeye haricen yönlendirilmeyen mutasyonlar, evrimsel algoritmayı çözüm uzayı içindeki yerel minimumlara takılmaktan kurtaramamaktadır. Örneğin CA-STG ile çözülmeye çalışılmış sorun budur. Tip güvenliğini gözetmekten dahi yoksun bir mutasyon operatörü çoğu zaman büyük sorunları gidermede küçük sorunlar oluşturacaktır (ör. semantik hataların giderilmesi sürecindeki ilk adımda sözdizimi hatalı yapıların oluşumu). Yerel minimumlara takılma sorununu seçilim algoritmasında çeşitliliği koruma yönündeki bir düzenlemeyle çözememenin bir nedeni bu sefer de çok daha fazla arama bütçesine (derinlik+dallanma) ihtiyaç duyabilecek soyların gelişimini engellemektir.

Özet olarak biyolojik evrimi ancak çok basit ve sınırlı sayıda mekanizmasıyla uygulamaya çalışan EA türevleri bu sınırlılığın imkan verdiği yeteneğe ulaşmıştır ve EA'yı yazılımsal sorunları çözmede genel geçer bir yöntem haline getirmeyeceği görülmüştür. Dolayısıyla EA'nın gelişiminde, biyolojik evrimin farklı mekanizmalarının incelenmesi ile daha uygun deneklerin oluşumunu sağlayacak mekanizmaların uyarlanması ilerlemeyi hızlandıracak gelişmelerin başında görülmüştür.

7.2 Uygulanabilirlik

GP (diğer) makine öğrenmesi yöntemleriyle karşılaştırıldığında iki özelliğiyle dikkat çeker. Birincisi, veri setinin sadece doğrulama amacıyla kullanmasıdır. İkincisi problemi çözmede gerekli olan alana özel bilgilerin kullanıcıdan haricen alınabilmesidir. GP teknik detayları GP'nin veya EA'nın iç işleyişi ile ilgili hiç bir bilgisi olmadan

kullanıcıdan soyutlamaya oldukça müsaittir. Bu niteliğiyle GP'nin özgün sorunları çözmede (diğer) makine öğrenmesi yöntemlerinden öne çıkacağı öngörülmektedir.

Dolayısıyla GP'nin kullanımı organizasyonel sorunların çalışanlar/geliştiriciler tarafından temel problemler, algoritmalar ve veri yapıları üzerinden ifadeleştirilmesi gereksinimini azaltır. Çünkü çalışanların yazılımsal sorunları çözmek için tek ihtiyacı zaten var olan (olması gereken) birim testlerini bir GP aracına vermeleridir.

Evrim pahalıdır. Çünkü veri setini, aramayı optimize etmek için kullanmak yerine sadece doğrulamak için kullanmak ve her zaman henüz ulaşılmamış olabilecek global minimumların araştırılması için aramayı dallandırmak toplam işlem sayısını ve süresini artırır. Ancak GP'yi organizasyonlar için uygulanabilir kılmak için maliyetini mümkün olanın ötesinde azaltmak gerekli olmayabilir. Çünkü görülmüştür ki işgücü maliyetleri dönemden döneme istihdamı imkansız kılacak kadar artabilirken, donanım maliyetleri dönemden döneme düşmekte, başarımları da yükselmektedir.

GP'nin uygulanabilirliğinin diğer metotlarla birlikte kullanılmasıyla yaygınlaşacağı öngörülebilir. Örneğin LLM'ler, yaygın karşılaşılan sorunlara çözüm üretmekte oldukça hızlıdır ancak özgün sorunları çözmekte bir seçenek değildirler. Organizasyonların karşılaştığı özgün sorunların, araçlar, yöntemler ve gereksinimlerdeki dönemsel, ön görülemez değişimlerden kaynaklandığı; ötesindeyse her dönem geliştirilen işin dönemin imkanları ve gereklilikleri içerisinde yaygın sorunlarla ifade edildiği ve yaygın aşina olunan, endüstri standardı kabul edilebilecek çözümlerle giderildiği dolayısıyla özgünlüğün ancak kuşbakışı var olduğu, özgünlüğün bileşenlerde ve detaylarda farklı derinliklerde sonlandığı görülebilir.

Çözümü AST üzerinde rastgele, asgari değişikliklerle bulmak yerine; algoritmalar, veri yapıları, tasarım desenleri seviyesindeki rastgele mutasyonlarla aramayla; yani çözümü bir üst organizasyon seviyesinde adımlarla aramayla; GP'yi sadece, çözümün bileşenlerinde özgünlüğün yitirildiği derinliğe kadar tercih ederek uygulanabilirliğin

yükseltilebileceği düşünülebilir. Ancak bu tür mutasyonlar sırasında dilin tip güvenliği dahil tüm kurallarını gözeten ve sorunları arama başlatmak gerektirmeksizin çözen bir GP'nin geliştirimi oldukça fazla iş gerektirmektedir.

Problem seçiciliği ve zaman maliyeti zafiyetleri olan GP'nin kullanımı yaygınlaştıracak gelişmenin, onu sadece donanım maliyetinin alternatiflerine kıyasla düşük kaldığı sorunlarda tercih edip diğer durumlarda daha ucuz yöntemleri kullanan ve EA'nın iç işleyişini kullanıcıdan tamamen soyutlayan bir aracın ortaya çıkması ve organizasyonlara sunulması olacağı ön görülmektedir.

8. ÖNERİLER

Çalışma süresinin azaltılması ve sürecin hızlandırılması için faydalı olabilecek pek çok yöntem fark edilmiştir. Yöntemlerden bazıları arama uzayını daraltmaya yönelikken, dağıtık ölçümleme kullanıcıya yansıtılan sürenin kısaltılmasına yöneliktir.

8.1 Kurallara Uygun AST Üretimi

STGP'nin ifade ağaçlarında uygulanışı bir operatör olan üst düğümün her bir parametresine o parametrenin kabul ettiği değer tipinde değer üreten bir düğümün bağlanması garantilenerek sağlanır (Montana 1995). Örneğin iki Boole değeri üzerinde mantıksal işlem yapan bir düğüme iki adet her biri aritmetik işlem yapıp tam sayı döndüren düğüm bağlanamaz. STGP'nin uygulanması sayesinde hatalı dizime sahip deneklerden arındırılarak arama uzayı daraltılır. Bu sayede süreç kısalır.

GP ifade ağaçları yerine AST ile kullanıldığında, STGP'nin doğrudan uygulanışı üst-alt düğüm tip eşleşmesini ifade, deyim, belirtim ve bildirim gibi düğüm tipi-sınıfları için gözetir. Örneğin bir IfStmt düğümünün koşul parametresi için ifade tip sınıfında bir tipten bir düğüm üretilerek STGP yeterliliği sağlanabilir. Ancak bu doğrudan uyarlama alt düğümlerin incelenmesini gerektirecek bir tip eşleşmesi kontrolünden yoksundur. Örneğin iki parametre alan ve tam sayı, floating point, string veya boolean değer üreten operatörlerin tamamı döndürdükleri değerlerin tiplerinin farklılığına karşın bir BinaryExpr tipinde düğüm ile AST içinde temsil edilir. Ancak tam sayı döndüren bir BinaryExpr'ının IfStmt koşulu olarak geçerliliği yoktur. Dolayısıyla GP'nin AST ile çalıştırılması STGP'nin alt-ağaçları inceleyecek gelişmiş bir versiyonunu kullanarak arama uzayını daha da daraltmayı mümkün kılar.

8.2 Türleştirme

NEAT (Stanley ve Miikkulainen 2002) yönteminde uygulanan türleştirme çaprazlamaların her zaman aynı tür içinden iki denek arasında gerçekleşmesini sağlar. Bu sayede ortak noktası çok az olan dolayısıyla parçaları diğeri için anlam ifade etmeyen iki alternatif çözüm yönteminin arasında gerçekleşebilecek görece faydasız çaprazlamaların önüne geçilir. Ayrıca tür içindeki deneklerin topolojileri benzer olduğundan çaprazlama için sınırların seçimi ve diğer denekle hizalanmaları kolaylaşır. Deneklerin türlerine karar vermek için bir nesildeki tüm deneklerin yapıları tür için referans seçilen deneklerle karşılaştırarak topolojik benzerlikleri ortaya çıkartılır. Yöntem, benzerliği yüksek olan denekleri aynı tür içinde toplar. Neticede türleştirme ile çaprazlamaları daha anlamlı ve kolay hale getirmek hedeflenir.

Katmanlı arama türleştirmenin uygulanmasını zorlaştırmaz. Aksine baskı, sözdizimi, çalışma zamanı hataları gibi teknik hatalara sahip ara-ürünleri aday uzayından ayırması nedeniyle türleştirmenin uygulanacağı popülasyonu yalınlaştırır. Aday uzayındaki popülasyon sadece kullanıcının sağladığı kriterlere uygunluğuyla değerlendirilen deneklerden oluşur hale gelir. Dolayısıyla nüfus daralır ve ölçeklenebilirlik yükselir.

8.3 Dağıtık Ölçümleme

Genetik programlama ile kod sentezi problemi dağıtımlı bilişime oldukça uygundur (Langdon 2023). Çünkü sürecin en yüksek zaman gereksinime sahip adımları, birbirinden bağımsız olarak gerçekleştirilebilir olan havuzdaki yeni adayların derlenmesi ve testlere tabi tutulması adımlarıdır. Bu adımların m adet sunucuda gerçekleştirilmesi kullanıcının sonuç bekleme süresini önemli ölçüde azalmasını sağlayacaktır. Üstelik m kullanıcısı olan bir hizmet için m sunucu alındığında/ kiralandığında, sunucu sayısının artırılması maliyeti doğrusal artırmayacaktır. Çünkü her ilave sunucuyla, bir sunucunun bir kullanıcı için kullanılma süresi kısalır.

Donanımın saatlik kiralandığı sağlayacılarla çalışıldığı durumda, ortalama maliyet ve ortalama çözüm süresi birlikte azalır.

8.4 Aksaklık Bölgesini Daraltma

Örneğin 10 adet değer karşılaştırması içeren bir birim-testinde 2 denek farklı 2 karşılaştırmada başarısız oluyorsa; çözümün bileşenleri farklı deneklerde mevcut ancak bir arada bulunmuyor olarak yorumlanabilir (istisnalar mevcut olabilir). Bu iki denek üzerinde gerçekleştirilecek bir sonraki çaprazlama, alt-ağaç seçimi sırasında diğer denekte farklılığı oluşturan bölgeleri önceliklendirme usulüyle sağlanarak evrimsel süreç hızlandırılabilir. Koddaki (dolayısıyla AST'deki) hangi bölgenin (if/while) farklı sonuca etki ettiği bilgisine ulaşmak için resmi Go hata ayıklayıcısı (ve pek çok dildeki hata ayıklayıcısı) tarafından sağlanan yürütme patikası bilgisi kullanılabilir. Bu tür bir iyileştirmenin bir diğer gereksinimi karşılaştırma sonuçlarını tek bir başarım değeri oluşturmak için birleştirmemek yani ayrı olarak saklamak ve denekler arasında sadece aynı değer karşılaştırması sonuçlarını karşılaştırmaktır.

Yürütme patikasına veya test kapsamına benzer bir bilginin GP içinde kullanılışına bir örnek Weimer vd. 2009 çalışmasında görülmüştür. Çalışmada kod tamirin aranacağı bölgeyi daraltmak ve sorunları çözerken çalışan işlevleri bozmamak için fonksiyon içindeki deyimlerin pozitif ve negatif sonuçlar beklenen durumların sınamalarındaki yürütümü sırasında ziyaret edilme durumundan yararlanılmıştır. Başarılı pozitif sınama örnek durumları sırasında ziyaret edilen deyimlerin ağırlığı 0.01 atanır. Hatayla sonuçlanması beklenen ancak sonuçlanmayan negatif örnek durumların yürütümü sırasında sadece tek bir kez ziyaret edilen deyimlerin ağırlığı artırılır.

8.5 Çok Karşılaştırmaya Sahip Birim Testlerinde Aksaklık Bölgesini Daraltma

Verilen birim testi çok sayıda değer karşılaştırması içerdiğinde kodda düzeltme aranan bölgenin daraltılması karşılaştırma başına türleşme sağlayarak denenebilir. Çünkü özel

kontrol akışından dolayı her bir karşılaştırma tarafından beklenen değerin üretiminin aynı deyimler tarafından gerçekleştirilmesi beklenmez. Bir karşılaştırma sonucunun iyileştirilmesi gereken arama sürecinin bulguları diğerleri için başarım düşürücü dahi olabilir.

8.6 LLM'ler Tarafından Üretilmiş Kodları Sürece Katma

Bir denekten bir nesilde oluşturulabilecek yeni deneklerin çeşitliliği ve sayısı düşünüldüğünde, 10'larca veya 100'lerce nesil doğru değişikliklerin seçilmesiyle ortaya çıkacak gerekecek iyileştirmelerin üretimini her nesilde birer sorgu ile elde etmek mümkündür. Bu sorgular bazen hatalı deneklerin kodundan önceye eklenecek "bu kodu tamir et" veya "alternatif çözümler üret" gibi ifadeler olabilir.

8.7 Birim Testlerinin Dönüştürümü

Standart test dosyalarını girdi olarak kullanamamak (Şekil 5.2, 5.3), kullanıcıların aracı denemeye başlamadan önce mevcut testlerini uyumlu hale getirmelerini gerektirmektedir. Ayrıca özel bir arayüzün kullanımı kısa bir eğitim süreci gerektirecektir. Bu durum başlangıç maliyetini artırmakla birlikte, kullanıcıların denemeyi ertelemesine de neden olabilir. Eğitim ve düzenlemeden kaynaklı yeniliği benimsemenin maliyetini düşürebilmek için resmi test paketi ve operatör bazlı karşılaştırmalarla düzenlenmiş test fonksiyonlarını AST düzenlemesiyle operasyona uygun formata dönüştürmek düşünülebilir.

Fonksiyon bazlı karşılaştırmaların kod içindeki konumunun tespiti için AST üzerinde DFS uygulanabilir. Bu tür karşılaştırmaları oluşturmak için sıklıkla kullanılan 3. taraf paketlerin fonksiyon ismi belirlidir. Gezinti sırasında bu fonksiyonlara yapılan çağrılar tespit edilip özel karşılaştırma fonksiyonu ile değiştirilebilir.

8.8 Sanallaştırma

Sanallaştırılmış test ortamının pek çok faydası öngörülmektedir. Bunlardan biri tüm sistemi test başına bir kere kullanarak, yan etkilere sahip birimlerin testini mümkün kılmaktır. Ayrıca ana sistemin güvenliğini etkilemeyeceği için tüm kütüphanelerin (os, exec dahil) aramaya katılmasını mümkün kılacaktır. Sanallaştırmanın denek başına oluşturulması ölçümleme süresini artıracağı için uygulanabilirliğin kaybedilmemesi için önlemler alınmalıdır.

9. SONUÇ

Genetik programlamanın mantıksal ifadelerin üretimi için kullanıldığı 90'lardan bu yana fonksiyonel dillere uyarlanması, arama uzayını daraltma amacıyla kod tamiri için kullanımı, AST'nin ifade ağaçı yerine kullanımı ve veri yapılarını kullanır hale gelmesiyle geçen süreç boyunca literatürde paylaşılan çalışmalar incelenmiştir. GP'nin gelecekte faydalı olabileceği gözde alanlardan biri olan zorunlu ve derlemeli dillerde kod sentezine uyarlanabilmesi için çözülmesi gereken sorunlar aktarılmıştır. Bu sorunlar için uygulanabilirliği yüksek yöntemler tasarlanmış ve paylaşılmıştır.

GP'nin nihayetinde organizasyonlar tarafından çalışanlara yardımcı bir araç olarak görülmesinin önündeki engeller olan yüksek maliyet ve problem seçiciliğinin giderilmesi için çözülmesi gereken sorunlar özetlenmiştir. Bu hedef doğrultusunda GP'nin (ve EA'nın) gelecek vaat ettiği görülmüştür. GP geliştirmeye açıktır, ve potansiyelinin çok küçük bir kısmı henüz açığa çıkmıştır.

Sonuç olarak; ilk 30 yılında paylaşılan çalışmalarla GP'nin çoğu yazılım sorunu için mevcut yöntemler olan programlama ve makine öğrenmesiyle rekabet edebilecek, özgün yazılım sorunlarında ise onları geçebilecek potansiyelde bir çözüm adayı olduğu ortaya konmuştur. Beklenen donanımsal gelişimlerle birlikte dezavantajlarının etkisinin azalacağı bu sayede alanda gerçekleşecek çalışmaların hızlanacağı öngörülmektedir.

Bu çalışmanın GP'nin popülaritesini ve topluma sağladığı faydayı potansiyeline çıkarması yolunda gerçekleşmesi gereken ve beklenen çalışmalara yol gösterici olması; dolayısıyla GP'nin gelişimi hızlandırması beklenmektedir.

KAYNAKLAR

Agapitos, A., O'Neill, M., Kattan, A.J., Lucas, S.M. 2017. Recursion in tree-based genetic programming. Genetic Programming and Evolvable Machines, 18, 149-183.

Anonymous. 2019. Web sitesi: https://stackoverflow.com/questions/556405/what-do-real-user-and-sys-mean-in-the-output-of-time1/556411#556411, Erişim tarihi 16.08.2023

Anonymous. 2022. Web sitesi: https://www.csoonline.com/article/567531/the-biggest-data-breach-fines-penalties-and-settlements-so-far.html Erişim tarihi: 28.08.2023

Anonymous. 2023. Evolution Strategy. Web sitesi: https://en.wikipedia.org/wiki/Evolution_strategy

Banzhaf, W., Nordin, P., Keller, R., Francone, F. 1998. Genetic Programming: An Introduction on the Automatic Evolution of computer programs and its Applications.

Choma, J., Guerra, E.M., Silva, T.S. 2018. Developers' Initial Perceptions on TDD Practice: A Thematic Analysis with Distinct Domains and Languages. International Conference on Agile Software Development.

Črepinšek, M., Liu, S., Mernik, M. 2013. Exploration and Exploitation in Evolutionary Algorithms: A Survey. ACM Computing Surveys. 45. Article 35. 10.1145/2480741.2480752.

Fernandes, M.C., de França, F.O., Francesquini, E. 2023. HOTGP - Higher-Order Typed Genetic Programming. ArXiv, abs/2304.03200.

Grudniewski, P.A., Sobey, A.J. 2019. Behaviour of Multi-Level Selection Genetic Algorithm (MLSGA) using different individual-level selection mechanisms. Swarm Evol. Comput., 44, 852-862.

Ha, D. 2017. A Visual Guide to Evolution Strategies. Web sitesi: https://blog.otoro.net/2017/10/29/visual-evolution-strategies/, Erişim Tarihi: 06.01.2021

Helmuth, T., Kelly, P. 2021. PSB2: The Second Program Syn-thesis Benchmark Suite. In 2021 Genetic and Evolutionary Computation Conference (GECCO '21), July 10–14, 2021, Lille, France. ACM, New York, NY,USA, 10 pages. https://doi.org/10.1145/3449639.3459285

Hooper, D. C., Flann, N. S., Fuller, S. R. 1997. Recombinative hillclimbing: A stronger search method for genetic programming. In Koza, J. R., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M., Iba, H., and Riolo, R. R., editors, Genetic Programming 1997: Proceedings of the Second Annual Conference, pages 174{179. San Francisco, CA: Morgan Kaufmann.

Hornby, G.S., Globus, A; Linden, D.S ve Lohn, J.D. 2006 Automated antenna design with evolutionary algorithms. American Institute of Aeronautics and Astronautics, https://www.researchgate.net/publication/228909002_Automated_Antenna_Design_with_Evolutionary_Algorithms

Human-Competitive "Human-Competitive Awards 2004 - Present", Web sitesi: https://human-competitive.org/awards, Erişim tarihi: 09.08.2023

Illanes, V. ve Bergel, A. 2021. Generating Object-Oriented Source Code Using Genetic Programming. 2021 IEEE/ACM International Workshop on Genetic Improvement (GI), 45-50.

Jackson, D. 2009. 2009 IEEE Congress on Evolutionary Computation içindeki “Self-adaptive focusing of evolutionary effort in hierarchical genetic programming”, 1821-28.

Koza, J.R. 1992a. Genetic Programming içindeki "Hierarchical Automatic Function Definition". FOGA.

Koza, J.R. 1992b. "Little LISP" Computer Code for Genetic Programming as Contained in 1992 Book Genetic Programming (Koza 1992) http://www.genetic-programming.org/gplittlelisp.html

Koza, J.R. 2010. Human-competitive results produced by genetic programming. _Genet Program Evolvable Mach_ **11**, 251-284 https://doi.org/10.1007/s10710-010-9112-3

Langdon, W.B. 2023. Jaws 30. Genet Program Evolvable Mach 24, 19. https://doi.org/10.1007/s10710-023-09467-x

Le Goues, C., Dewey-Vogt, M., Forrest, S., Weimer, W. 2012b. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. 2012 34th International Conference on Software Engineering (ICSE), 3-13.

Le Goues, C., Nguyen, T., Forrest, S., & Weimer, W. 2012a. GenProg: A Generic Method for Automatic Software Repair. IEEE Transactions on Software Engineering, 38, 54-72.

Le Goues, C. 2021 GenProg: heuristic, GP-based automatic program repair for C. Web sitesi: https://github.com/squaresLab/genprog-code Erişim tarihi 17.10.2023

Le, X.D., Lo, D., & Le Goues, C. 2016. History Driven Program Repair. 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 1, 213-224.

Lohn, J.D., Hornby, G.S. ve Linden, D.S. 2004 An Evolved Antenna for Deployment on NASA's Space Technology 5 Mission. Genetic Programming Theory Practice 2004 Workshop (GPTP-2004), https://www.researchgate.net/publication/226096470_An_Evolved_Antenna_for_Deployment_on_Nasa%27s_Space_Technology_5_Mission

Luke, S., ve Panait, L. 2002. Fighting Bloat with Nonparametric Parsimony Pressure. Parallel Problem Solving from Nature.

Montana, D. 1995. Strongly Typed Genetic Programming, Evolutionary Computation, 3(2)15175.pdf, Erişim tarihi: 06.01.2021

Pantridge, E.R., Helmuth, T., Spector, L. 2022. Functional code building genetic programming. Proceedings of the Genetic and Evolutionary Computation Conference.

Poli, R., Langdon, W. B., McPhee, N. F. 2008. A field guide to genetic programming. http://www.gp-field-guide.org.uk, 2008. (With contributions by J. R. Koza).

Sipper M. 2019. Tiny Genetic Programming in Python. Web sitesi: https://github.com/moshesipper/tiny_gp, Erişim tarihi: 16.10.2023

Sloss, A.N., Gustafson, S.M. 2019. Evolutionary Algorithms Review. Genetic Programming Theory and Practice.

Stackoverflow.com. 2022. 2022 Developer Survey. Web Sitesi: https://survey.stackoverflow.co/2022#professional-developers-developer-experience, Erişim tarihi: 01.07.2023

Stackoverflow.com. 2023. 2023 Developer Survey. Web Sitesi: https://survey.stackoverflow.co/2023/#professional-developers-developer-experience, Erişim tarihi: 21.07.2023

Stanley K.O., Miikkulainen R.; Evolving Neural Networks through Augmenting Topologies. Evol Comput 2002; 10 (2): 99-127

Yıldırım, U. 2023. Web sitesi: https://github.com/ufukty/language-survey-review, Erişim tarihi: 19.08.2023

Yuan, Y. Banzhaf, W. 2017. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. IEEE Transactions on Software Engineering, 46, 1040-1067.

Wagner, G.P., & Altenberg, L. (1996). Perspective: Complex Adaptations And The Evolution Of Evolvability. Evolution, 50.

Weimer, W., Nguyen, T., Le Goues, C., Forrest, S. 2009. Automatically finding patches using genetic programming. 2009 IEEE 31st International Conference on Software Engineering, 364-374.

EK 1 TERİMLER DİZİNİ

Terimler parçadan bütüne sıralanmıştır:

Denek (Subject) Rastgele düğümlerle üretilmiş veya bunların çaprazlanması veya mutasyonlara uğratılmasıyla evrimleşmiş, bir nesne. AST, kod, program, aday veya çözüm statülerinden birinde olabilir.

AST Denek statülerinden ilkidir. Bu statüdeki denekler basılabilirse Kod statüsüne yükselirler.

Kod (Code) Denek statülerinin ikincisidir. Bu statüdeki denekler derlenebilirse Program statüsüne yükselirler.

Program Denek statülerinin üçüncüsüdür. Bu statüdeki denekler sınanmalarını (değer karşılaştırmalarında başarısız olmak gibi durumlar hariç, erken sonlanmaya yol açan türde) hata vermeden tamamladıklarında Aday statüsüne yükselirler.

Aday (Candidate) Denek statülerinin dördüncüsüdür. Basılabilmiş, derlenebilmiş, sınanmış deneklerdir.

Çözüm (Solution) Denek statülerinini beşincisidir. Tüm değer karşılaştırmalarında beklenen sonuçları üretebilmiş deneklerdir.

Nesil (Generation) Seçilim-türetim adımlarının birer kez yürütülmesiyle ilerleyen bir değer.

Havuz (Pool) Deneklerin kümesi. İçerdiği deneklerde veya deneklerin yapılarında nesilden nesile değişiklik gözlenir. Kaç denek içerdiği geliştirici veya kullanıcı tarafından belirlenebilir.

Genetik operasyon (Genetic operation) Bir deneğin yapısında gerçekleştirilen bir değişimle yeni bir denek oluşturulan işlemler: çaprazlamalar ve mutasyonlar.

Türetim	Seçilimden sonra havuzda kalan deneklerden genetik operasyonlarla yeni deneklerin türetilmesidir.
Birim-testi	Programlamada bir fonksiyonun çeşitli girdilere karşı üretmesi gereken değerleri ürettiğini doğrulayan, geliştiriciler tarafından hazırlanan fonksiyonlara denir. Çalışmada başarım fonksiyonu olarak kullanılır.
Başarım fonksiyonu	Seçilim aşamasında daha iyi denekleri belirlemek için kullanılan fonksiyondur. Girdi olarak aldığı denek bir programın (varyasyonun) istenilene yakınlığını (veya uzaklığını) döndürür.
Seçilim	Havuzdaki deneklerden başarım fonksiyonuna göre daha uyumsuz olanlarının, diğerlerinden türetilecek yeni deneklere yer açmak için elendiği aşamadır.
Evrim	Seçilim-türetim evrelerinin nesiller boyu tekrarıyla, kısıtlara daha uyumlu karmaşık yapıları, çözümün formuna dair ön farkındalığa ihtiyaç duymaksızın elde etmeyi sağlayan süreç.

EK 2 BASİT MUTASYON ÖRNEKLERİ

STG'nin sadece AST seviyesinde tip güvenliği gözeten ve bağlamın farkında olmayan yani ST-ASTGP olmayan basit versiyonu ile tek bir ortak atadan üretilen 200 yavru arasından basım hatalarına sahip olmayan (kod statüsündeki) 28 yavrunun listesi:

Atanın kod temsili:

```
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    walkHelper(root, []ast.Node{}, []int{}, callback)
}
```

Kod karşılığı olan yavrular:

```
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    walkHelper((var870.), root, []ast.Node{}, []int{}, callback)
}
---
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    walkHelper(root, []ast.Node{}, []int{}, callback, 0)
}
---
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    select {
    goto BranchLabel4
    }
    walkHelper(root, []ast.Node{}, []int{}, callback)
}
---
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    walkHelper(root, [var2610[(): 0)]]ast.Node{}, []int{}, callback)
}
---
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    walkHelper(root, []ast.Node{}, ...var5541, []int{}, callback)
}
---
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    walkHelper(root, &var8657, []ast.Node{}, []int{}, callback)
}
---
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    walkHelper(root, []ast.Node{}, [...*]int{}, callback)
}
---
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    walkHelper(root, []ast.Node{}, var12003, []int{}, callback)
}
---
```

```
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    walkHelper(root, []ast.Node{}, [var12392.]int{}, callback)
}
---
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    walkHelper(root, []ast.Node{}, [var12393: *1]int{}, callback)
}
---
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    walkHelper(root, []ast.Node{}, [var15508]int{}, callback)
}
---
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    walkHelper(root, [*var15761]ast.Node{}, []int{}, callback)
}
---
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    goto BranchLabel26
    walkHelper(root, []ast.Node{}, []int{}, callback)
}
---
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    goto BranchLabel82
    walkHelper(root, []ast.Node{}, []int{}, callback)
}
---
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    walkHelper(root, []ast.Node{}, []int{}, [0.794055288192076],
callback)
}
---
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    walkHelper(root, []ast.Node{}, []int{}, callback,
(0.7387086491617374))
}
---
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    walkHelper(root, []ast.Node{}, (......*var25839), []int{},
callback)
}
---
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    walkHelper(root, []ast.Node{}, [&1]int{}, callback)
}
---
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    walkHelper(root, (0), []ast.Node{}, []int{}, callback)
}
---
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    walkHelper(root, []ast.Node{}, []int{}, callback, )
}
---
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    walkHelper(root, q, []ast.Node{}, []int{}, callback)
}
---
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    break BranchLabel51
    walkHelper(root, []ast.Node{}, []int{}, callback)
}
```

```
---
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    walkHelper(var31120, root, []ast.Node{}, []int{}, callback)
}
---
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    walkHelper(var31121[1]., root, []ast.Node{}, []int{}, callback)
}
---
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    walkHelper(0, root, []ast.Node{}, []int{}, callback)
}
---
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    walkHelper(root, []ast.Node{}, []int{}, callback, 0)
}
---
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    walkHelper(root, []ast.Node{}, [var35332]int{}, callback)
}
---
func WalkWithNils(root ast.Node, callback WalkCallbackFunction) {
    walkHelper(root, []ast.Node{}, []int{}, var35364, callback)
}
```