Mühendislik Bilimleri ve Tasarım Dergisi
8(5), 113-125, 2020
e-ISSN: 1308-6693
Araştırma Makalesi

Journal of Engineering Sciences and Design
DOI: 10.21923/jesd.828809
Research Article

Özel Sayı: Uluslararası Mühendislikte Yapay Zeka ve Uygulamalı Matematik Konferansı (UMYMK 2020)
Special Issue: International Conference on Artificial Intelligence and Applied Mathematics in Engineering (ICAIAME 2020)

EKRAN ÇÖZÜNÜRLÜĞÜNE DUYARLI BİR AKILLI ARAYÜZ YERLEŞTİRME YAKLAŞIMI

Barış Çelik[1*], Burkay Genç[2]

[1] Hacettepe University, Informatics Institute, Department of Computer Animation and Game Technologies, Ankara, Türkiye
[2] Hacettepe University, Engineering Faculty, Department of Computer Engineering, Ankara, Türkiye

Anahtar Kelimeler

Değişken, Çözünürlük, Akıllı Yerleştirme, Kullanıcı Arayüzü, Duyarlı Tasarım.

Özet

Bu makalede ilişkisel olarak tanımlanan bir kullanıcı arayüzünün akıllı ve dinamik bir yaklaşımla, gerçek zamanlı olarak değişken ekran çözünürlüklerine en iyi şekilde adapte olmasını sağlayacak bir algoritma sunuyoruz. Kullanıcı arayüzleri bilgisayarlar ve kullanıcılar arasındaki etkileşimi sağlayan yazılım ürünleridir. Bu arayüzlerin ortaya çıkartılmasında tasarımcılar ile yazılımcılara düşen görevler vardır. Günümüzde değişik cihazların desteklediği çok farklı ekran çözünürlüklerinin kullanımda olmasıyla birlikte geliştirilen uygulamalarında her ekran çözünürlüğünde başarıyla çalışması beklenmektedir. Fakat, mevcut durumda tüm farklı çözünürlükler için tasarımcılar ve yazılımcılar farklı arayüzler geliştirmekte ve bu arayüzlerin hem ilk geliştirimi hem de devam eden süreçte bakım ve güncellenmesi ciddi bir külfet getirmektedir. Bu problemin çözümü adına literatürde farklı yaklaşımlar önerilmiş olmakla birlikte, bu yaklaşımlar tasarım aşamasında harcanan eforu azaltmaya yönelik, gerçek zamanlı çalışmayan yaklaşımlardır. Bu çalışmada biz tasarımcı ve yazılımcıların üzerinden bu yükü alacak, uygulamanın geliştirilmesi esnasında bir defa ve basit bir ilişkisel modelle tanımlanacak bir veri yapısını ve bu yapıyı gerçek zamanda işleyerek verilen ekran çözünürlüğüne en uygun arayüze dönüştüren gerçek zamanlı bir yaklaşımı sunuyoruz. Kullandığımız veri yapısının hazırlanması son derece kolay olduğu gibi, tek bir arayüz tasarımından daha kısa bir zamanda hazırlanabilmektedir. Uygulamanın çalışması esnasında arayüzün oluşturulması da saniyenin altında gerçekleşmekte, gerçek zamanlı yeniden boyutlandırma işlemleri esnasında dahi arayüzde gecikme yaşanmamaktadır.

A SCREEN RESOLUTION SENSITIVE INTELLIGENT INTERFACE LAYOUT APPROACH

Keywords

Variable, Resolution, Smart Layout, User Interface, Responsive Design.

Abstract

In this article, we present an algorithm that will enable a user interface defined as relational, with a smart and dynamic approach, to best adapt to variable screen resolutions in real time. User interfaces are design and software products that enable interaction between computers and users. Both designers and developers work on creating these interfaces. Nowadays every software is expected to run on every resolution; and developers create separate interfaces for different resolutions. The maintenance of these interfaces both in the initial development and in the ongoing process is a serious burden. Different approaches are suggested in the literature for the solution to this problem; these approaches do not work in real time and are for reducing the effort spent in the design phase. In this study, we present a real-time approach that takes this burden off the developers. We propose a simple relational model that is defined once during the development phase, and this model is processed in real time and transforms into the most suitable interface for the given screen resolution. The data structure can be prepared in a shorter time than a single interface design. In runtime, the creation of the interface takes place under a second.

Alıntı / Cite

Çelik, B., Genç, B. (2020). A Screen Resolution Sensitive Intelligent Interface Layout Approach, Journal of Engineering Sciences and Design, 8(5), 113-125.

[*] İlgili yazar / Corresponding author: baris_celik@hacettepe.edu.tr, +90-538-870-9794

Yazar Kimliği / Author ID (ORCID Number)	Makale Süreci / Article Process	
B. Çelik, 0000-0003-2506-3676	**Başvuru Tarihi / Submission Date**	20.11.2020
B. Genç, 0000-0001-5134-1487	**Revizyon Tarihi / Revision Date**	27.12.2020
	Kabul Tarihi / Accepted Date	28.12.2020
	Yayım Tarihi / Published Date	29.12.2020

1. Giriş

Günümüzde üretilen yazılımlar pek çok farklı cihaz ve platform üzerinde çalışabilmek üzerine tasarlanmaktadır. Ancak tüm cihaz ve platformları destekleyebilmek için, her birine özgü ayrı bir arayüz tasarımı yapılması gerekmektedir. Farklı ekran boyut ve çözünürlüğüne sahip yüzlerce telefon modeli mevcuttur. Roudaki vd., (2014) web deneyimini mobil cihazlara aktarabilmek için çeşitli yaklaşımlar özetlemiştir. Ayrıca bu tecrübeyi otomatize etmeye çalışan yazılım kütüphaneleri, bu işi kolaylaştırmak adına otomatik yerleştirme algoritmaları sunmaktadır. Ancak bu algoritmalar oldukça basittir ve işin büyük kısmını tasarımcı ve yazılımcılara yaptırmaktadır. Geliştirdiğimiz algoritma yazılımcı ve tasarımcılara düşecek iş miktarını minimize edecektir.

Geliştirdiğimiz yöntem, yazılımcıdan minimal bir bilgi alır. Bu bilgi her bir bileşenin minimum ve maksimum büyüklük bilgisi ile hangi bileşenlerin "alakalı" olduğu bilgisidir. Alakalı olmak geometrik değil soyut bir bilgidir ve bu noktada yazılımcının spesifik bir yerleştirme algoritması seçmesini gerektirmez. Ardından, arayüz ekrana çizileceği zaman, yazılım geliştirme esnasında alınmış olan bilgi kullanılarak akıllı bir algoritma vasıtasıyla tüm bileşenlerin genişliği, yüksekliği ve pozisyonu hesaplanır ve arayüz otomatik olarak oluşturulmuş olur. Bunu yapmak için algoritma tüm muhtemel yerleştirmelerin ağacını hızlı bir şekilde budayarak iyi bir çözüme ulaşır. Bu işlemin saniyenin çok altında bir sürede yapılması hedef olduğundan göreceli küçük arayüzlerde tam en iyileme, çok karmaşık arayüzlerde ise en iyiye çok yakın sonuçlar elde edilmektedir. Böylelikle geliştiriciler sürekli farklı çözünürlükler için sayfalarını tekrardan düzenlemek zorunda kalmazlar.

Geliştirdiğimiz algoritmanın yaptığımız incelemelerde birebir bir rakibi yoktur. Örneğin Java programlama dilinde yerleştirme işlemleri LayoutManager'lar tarafından yapılmaktadır ancak Java dilindeki LayoutManager'ların hiçbirisi değişen ekran çözünürlüklerine göre bileşenlerin göreceli pozisyonlarına dokunmamakta, yalnızca büyüklüklerini değiştirmektedir. Benzer şekilde Android mobil işletim sisteminde de LinearLayout, RelativeLayout gibi yerleştirme algoritmaları kullanılmakta, ancak bunlar yalnızca bileşenlerin büyüklükleriyle ilgilenmektedir. Web tasarımında ise işler daha da kötüdür. Değişken ekran çözünürlüklerine göre arayüzü değiştirmek çok az sayıda web sayfasında gördüğümüz bir uygulamadır. Genellikle web sayfaları belli bir genişlik önkabul edilerek yapılmakta, bu genişliğin üzerindeki ekran boyutlarında sayfanın büyük kısmı boş kalmakta, daha küçük genişlikteki ekranlarda ise içerik ekrana sığmamaktadır. Şirketler mobil cihazların günlük hayattaki kullanımı arttıkça alternatif olarak mobil cihaz boyutlarını kapsayacak ek tasarımlar yaratmaya başlamıştır. Fakat bu tasarımlar oldukça maliyetli olduğu gibi teknoloji ilerledikçe oluşan yeni ölçeklendirmeleri kapsamak oldukça güç olmuştur. Zamanla bu probleme alternatif çözümler getirilmeye çalışılmıştır. Bunların en çok kullanılanı Bootstrap ismindeki Twitter tarafından geliştirilen frameworktür. Bootstrap tanımladığı "Responsive Design / Duyarlı Tasarım" mantığıyla farklı ekran çözünürlüklerinde arayüzü değiştirmeye çalışmaktadır. Ancak Bootstrap akıllı bir sistem değildir. Tasarım aşamasında belirlenen bazı koşullara göre arayüzü değiştirmektedir. "X çözünürlüğünün altında bir çözünürlükte Y bileşeninin genişliğini azalt" gibi bir örnek verilebilir. Bootstrap ile beraber, artık tasarım yapılırken her ekran genişliğinde göze hoş geleceği düşünülen ve her ölçeklendirmeye adapte olabilen tasarımlar üretilmeye başlanmıştır. Fakat farklı çözünürlüklerde otomatik olarak kendini boyutlandıran bu sistemler, çözünürlük farkı büyüdükçe göze batan hataları beraberinde getirmektedir. Geliştirme sürecinde ise önkoşullar geliştirici tarafından belirlenip, farklı çözünürlüklerde tekrar tekrar test edilmektedir. Onca çabaya rağmen olası hataları ve bozuklukları gözden kaçırmak ise oldukça mümkündür. Bizim önerdiğimiz algoritma ise önceden bu tarz koşullar oluşturmadan tamamen uygulamanın çalışması esnasında en iyi arayüz yerleştirmesine karar verecek ve uygulayacak akıllı bir algoritmadır. Başka benzer yaklaşımlar da bulunmakta, hatta firmalar kendi özgün yaklaşımlarını da getirmeye çalışmaktadır. Ancak tüm bu eforları kapsayacak, bilimsel bir yaklaşımın eksikliği görülmektedir.

Problemin tanımını akademik olarak araştırdığımızda ise, bu konuya benzer tüm yayınlar arayüz editörleri ve arayüz şablonları üzerindedir. Arayüz editörleri, bir arayüzün sadece kod yazarak değil görsel olarak da tasarlanarak oluşturulabilmesini sağlar. Bu editörler "sürükle-bırak" editörleri olarak da bilinir. Temel işlevler, bileşenleri bir araç çubuğundan sürükleyip uygulama ekranına bırakarak uygulama arayüzünü görsel olarak tasarlamaktır. Bu tür arayüz editörleri ile uygulamalar geliştirirken, normalde her bir elemanın kısıtlamaları ve boyut olarak ne kadar değiştikleri belirlenir. Literatürde bu tür editörlerin otomatik yerleştirme algoritmalarını görsel olarak kullanmasına izin veren yayınlar bulunmaktadır ve bu bölümde bahsi geçen yayınların çıkardığı çalışmalara değinilecektir.

2. Kaynak araştırması

Akıllı ortamlar, kullanıcı arayüzlerinin, çoğu zaman tasarım sırasında öngörülemeyen kullanım bağlamlarına dinamik olarak adapte olmasını gerektirir. Bu sorunu çözmek için, gerçek zamanlı arayüz modellerinin kullanılması önerilmiştir. (Sotted vd., 2006) Doğru modellemelerle arayüzü işleyip optimal sonuçlar bulan çalışmalar literatürde mevcuttur. Gajos ve Weld (2004), bizim bu çalışmada önerdiğimize benzer bir yapı önermiş ve bu yapı üzerinde kısıt çözücüler kullanarak tasarımcıdan alınan kısıtlara uygun bir arayüzün otomatik geliştirilmesini sağlamışlardır. Daha spesifik bir bölme yöntemi kullanılarak, çalışmamızdaki gibi arayüz elemanları bir topolojik yapıya dökülmekte ve bu yapıda veri tipine kadar detaya inilerek bir ağaç oluşturulmaktadır. Ağaçtaki uç birimler sayfada görünen arayüz elemanlarını temsil ederken ara birimler de ilişkileri temsil etmektedir. Bu çalışmada, bizim önerdiğimizden farklı olarak kısıt çözücüler kullanılmış, alternatif yerleştirmelerin de bulunulmasına ise odaklanılmamıştır. Roscher vd., (2011) ise topolojik bir yapı kurmaktadır fakat kurguladığımız yapıya en yakın yöntem olarak bir de yapısında her bir birimde pozisyon, boyut ve yatay veya dikeylik değerlerini barındırmaktadır. Çalışmada yapılan iş genel olarak sözel bir dille ifade edilmekte ve teorik detaylardan çok fazla bahsedilmemekte; bu da çalışmanın tekrarlanabilirliğinin önüne geçmektedir.

Literatürü incelediğimizde, benzer çalışmalar görmekle birlikte bu proje kapsamında tanımladığımız şekilde, gerçek zamanlı, akıllı bir yerleştirme algoritmasının henüz çalışılmadığını gözlemlemekteyiz. Literatürdeki benzer çalışmalar ağırlıklı olarak arayüz tasarımının kısıtlar üzerinden yapılması, sonra bu kısıtların bir optimizasyon kütüphanesi üzerinde çözülmesi ve çözümün statik olarak yazılımın içine gömülmesi esasına dayanmaktadır. Bu yüzden çalışmalar, ağırlıklı olarak arayüz tasarım modellerinin oluşturulması ve ortaya çıkan modellerin verimli bir şekilde çözülmesine odaklanmış ve çıktı olarak algoritmalar (Borning vd., 1997; Badros vd., 2001) ve araçlar (Hosobe, 2000; Hosobe, 2001) sağlamışlardır. Örneğin, Jamil vd. (2013), Kaczmarz algoritmasını arayüz tasarımında karşılaşılan eşitsizlik kısıtlarıyla çalışabilir hale getirmiştir. Yine Jamil ve Noreen (2014), kısıt çözücülerin arayüz tasarımında nasıl kullanılabileceğini araştırmıştır ve bu yöntemi geliştirecek bir önceliklendirilmiş gruplama kısıtları algoritması geliştirmiştir (Jamil vd., 2016). Bu algoritma, kısıtlamalar sisteminden art arda kısıtlar ekleyerek veya çıkararak çelişkisiz bir kısıtlama sistemi bulmaya çalışan bir algoritmadır. Borning vd. (1997) de benzer şekilde haber siteleri için arayüz kısıt modellerini verimli çözebilen dual simplex metodu tabanlı çözücüler üzerine çalışmışlardır. Jacobs vd., (2003) ise, grid tabanlı şablonlar tanımlayarak içeriği bu şablonlar ile eşleştirmişlerdir.

Jakob Nielsen (1994), ise araştırmasında kullanıcı deneyimi üzerine çalışmalar yapmıştır ve bu çalışmalarında sayfadaki elementler üzerinden giderek kullanılabilirliği yüksek arayüz tasarımı nasıl yapılmalıdır sorusunu cevaplamaya çalışmaktadır. Bizim çalışmamızda arayüz elemanlarının ekranda görünüp görünmemesi veya hangi arayüz bileşeni kullanarak görselleneceği bir seçenek değil, tasarımcının belirleyerek algoritmaya girdi olarak verdiği bir bilgi olduğu için, bu makale kapsamında kullanılabilirliğe değinmek mümkün görünmemektedir. Bununla birlikte, ileride algoritmalarımızı baz alan kullanılabilirliğin de optimize edildiği otomatik yerleştirme çalışmalarının yapılabilmesi mümkündür.

Editör tabanlı yaklaşımlar üzerinden ilerlemektense yeni bir bakış açısı kazandırmaya çalışan Jiang vd. (2019), ORC Layout adlı çalışmasıyla kullanılan standart layoutların harmanını çıkartarak hem Grid Layout'u hem de Flow Layout'u destekleyen bir yapı geliştirmiştir ve bu yapı sonuç üretmek için Satisfiability Modulo Theory, kısaca SMT denilen bir çözücü kullanmaktadır. Çalışmada ORC Layout'u düzenleyebilmek için bir de editör yazılmış ve bu editör üzerinden tasarım yapılarak bizim çalışmamızda hedeflediğimiz editör ve kısıt çözücü bağımsız bir yaklaşımdan uzaklaşılmıştır.

Zeidler vd., (2012), çalışmasında Auckland Layout Editor (Lutteroth vd., 2008) adlı programı kullanarak kısıt çözümleme üzerine çalışmıştır. Ancak bu yaklaşımlar tam olarak akıllı bir sistemi anlatmamakta, sadece tasarım aşamasında kullanılabilecek yöntemler sunmaktadırlar. Zeidler vd., (2013), başka bir çalışmada bunlara ek olarak, düzenleme operasyonlarına detaylı tanımlamalar yapmış, tasarım aşamasında sayfa elemanlarının birbirine karışmamasını sağlayan bir algoritma geliştirmiş ve yine tasarım aşamasında tekrardan boyutlandırılan bir elementin davranışını görebileceği bir eklenti geliştirmiştir. Geliştirmelerdeki asıl amacı ise düzenin her an korunur olmasıdır ve hiçbir şekilde iç içe giren elementleri barındırmamaktır. Kısıt çözümleyiciyi birden fazla seçenekle boğmamak için özel ve tek bir çözüm üretmeyi hedeflemiştir. 2017 yılında Zeidler vd., (2017) algoritmasını geliştirerek arayüz elemanları arasında parselleme çalışması yaparak çözümleyicilerin gücünü arttırmaya çalışmıştır. Fakat Zeidler bu üç çalışmanın sonunda hala tasarım aşamasında belirlenen kısıtlarla işlem yapabilmektedir ve bu yöntem sadece normalde yapılan işlemlere daha teknolojik bir yaklaşımdır. Bizim amacımız ise tasarım aşamasında sadece temel bilgiler alıp çalışma esnasında bu bilgiler ışığında tasarımı güncellemektir.

Benzer şekilde şablon tabanlı yaklaşımlar da, önceden belirlenen şablonlar ile içeriklerin birbirine eşlenmesini sağlamakta, böylece eşlenen içerik önceden tanımlanmış ve tasarlanmış şablona uygun şekilde çizilmektedir.

Örneğin, aynı içeriğin dergi, gazete, tabloid ortamlarında sunulması için önceden bu ortamlar için hazırlanmış şablonlar kullanılmaktadır. Mesela, Jacobs vd., (2003), grid tabanlı şablonlar tanımlayarak içeriği bu şablonlarla eşitlemiştir. Bu çalışma aslında günümüzde Bootstrap tarafından Web sayfalarının tasarlanmasında kullanılan yöntemle paralellik göstermektedir.

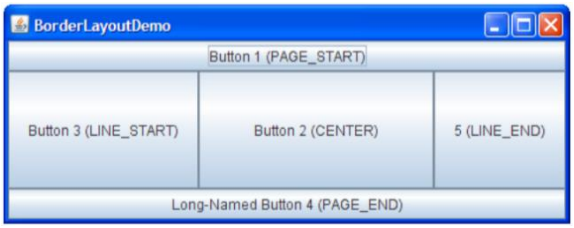
Bu yaklaşımların hepsi de Bootstrap ve CSS mantığı ile daha ilgilidir ve arayüzün tasarım esnasında göreceli olarak sabitlenmesi esasına dayanır. Bir başka deyişle, tasarımcı program çalışmadan önce arayüzdeki bileşenlerin hangi sırayla ve hangi doğrultuda (yatay-dikey-grid) yerleştirileceklerine karar verir. Bu bilgileri kullanarak bir kısıt modeli oluşturur ve bu modeli bir kısıt çözücüde çözer. Çözümü de uygulama arayüzüne yansıtır. Halbuki, bizim geliştirdiğimiz sistemde, tasarımcı sadece hangi bileşenlerin birbiriyle ilişkili olduğunu belirtecek, onların hangi sırada veya hangi doğrultuda ekranda görüneceğine karışmayacaktır. Bu tür kararlar, algoritmanın kendisi tarafından uygulamanın çalışması esnasında yapılmaktadır.

Bunun yanında, bir kullanıcı arayüzünün estetik ve kullanılabilirlik açısından değerlendirilmesi ve hatta sayısal olarak puanlanması üzerine yapılmış çalışmalar mevcuttur. Son dönemde yapılan kapsamlı çalışmalardan bir tanesi Pajusalu ve Maarja (2012) tarafından yapılmıştır. Pajusalu, bu çalışmada farklı web sitelerini incelemiş ve kullanıcılara bu web siteleri üzerinden anketler uygulayarak hangi web sitesinin neden beğenildiğini ortaya çıkarmaya çalışmıştır. Buanga ve Mbenza (2011) ise tezinde arayüz estetiğinin otomatik değerlendirilmesi üzerine çalışmış, bir arayüzün objektivist bakış açısıyla nasıl değerlendirilebileceğini ve hangi kriterler üzerinden nasıl notlanacağını incelemiştir. Fakat bu çalışmalar arayüzün yerleştirilmesiyle ilgili değil nasıl yerleştirildiğinin yorumlanmasıyla ilgilidir ve bu sebepten dolayı bu makalenin kapsamı dışındadır. Ancak bir sonraki aşamalarda ilerletmek açısından bir adım olarak görülebilir.

3. Metodoloji

3.1. Veri yapısı

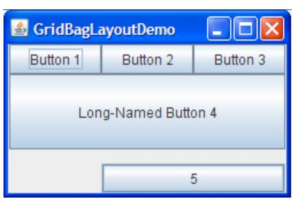
Algoritmanın temel hedeflerinden birisi tasarım ve kodlama aşamalarında yazılımcılardan istenecek bilginin en aza indirgenmesi ve yazılımcının mümkün olduğunca az karar vermesinin sağlanmasıdır. Mevcut yazılım dillerinde arayüz oluşturmak istediğinizde çok kritik bazı kararları vermeniz gerekir. Örneğin Java Swing kütüphanesi ile bir uygulama arayüzü geliştirmek istiyorsanız, öncelikle Java'nın (Zukowski, 1997) size sunduğu yerleştirme algoritmalarından hangilerini kullanacağınıza karar vermeniz gerekir. Örneğin BorderLayout bunlardan bir tanesidir ve ekranı üst, alt, sağ, sol ve merkez olmak üzere beş parçaya ayırmanıza ve her bir parçaya bir bileşen yerleştirmenize yarar.



Şekil 1. BorderLayout algoritması tarafından oluşturulan bir yerleştirme örneği

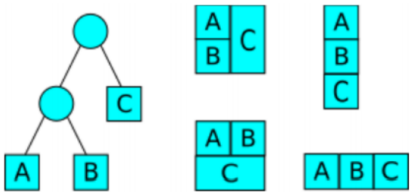
Genellikle uygulama veya ekran boyutu değiştirildiğinde, BorderLayout bu değişimi merkez bölmeyi büyüterek veya küçülterek tolere eder. Ancak ek olarak yazılımcı tercih edilen büyüklük bilgilerini de her bir bileşene atayabilir. Bu bilgiler gerekli görülürse BorderLayout algoritması tarafından bölmelerin büyüklüklerinin hesaplanmasında kullanılır.

Bir başka örnek GridBagLayout algoritmasıdır. Bu algoritmada ise ekran satır ve sütunlara bölünür ve bileşenler bu satır ve sütunlara yerleştirilir. Yazılımcının önceden her bir bileşenin hangi satır ve sütunları kapsayacağını belirlemesi gerekir. Bunun yanında da pek çok kısıt verilerek ekran görüntüsünün istenildiği gibi oluşturulması sağlanır.



Şekil 2. GridBagLayout algoritması tarafından oluşturulan bir düzen örneği

Geliştirdiğimiz algoritma ise sayfadaki bileşenlerin görelilik bilgilerini tutmak için bir ağaç yapısı kullanıyor. Yazılımcının arayüzde görmek istediği her bir bileşen bu ağaçta bir yaprak olarak yer alırken, ağacın dalları bileşenler arasındaki topolojik ilişkileri depoluyor. Her yaprak düğümü, kendisinin genişlik ve yükseklik aralıklarını ve atanan koordinatları ve atanan genişlik ve yükseklik değerlerini tutar. Her bir dal yalnızca topolojik bir ilişkiyi sembolize ederken, algoritma çözümlemeyi tamamladığında bu topolojik ilişki bir geometrik ilişkiye evrilecektir. Örnek olarak Şekil-3'ü inceleyelim.



Şekil 3. Solda topolojik ilişki ağacı, sağda dört olası yerleşim

Bu resimde 3 adet arayüz bileşeninin topolojik ilişkilerinin bir ağaçta depolanmasını görüyoruz. Bu ağaç bize, A ve B bileşenlerinin topolojik olarak bir ilişkisinin olduğunu, yani bir başka deyişle, bağlantılı olduklarını, C bileşeninin ise A ve B bileşenlerinin kümesi ile topolojik olarak ilişkili olduğunu söylüyor. Bu topolojik ilişkiler, 4 farklı geometrik ilişkiye evrilebiliyor. Bunlar aynı resimde sağ tarafta gösterilmektedir. C düğümü A ve B düğümlerinin ebeveyn düğümünün "sağında" olduğu için yerleştirme esnasında da C bileşeni A ve B bileşenlerinin sağında veya altında konumlanmaktadır. Bu ağacın yaratılışından çıkartılan ama yazılımcının direkt olarak vermediği bir bilgidir. Bu sebeple bu bilginin geçersiz olduğu yazılımcı tarafından özellikle belirtilirse C bileşeni A ve B bileşenlerinin solunda ve üstünde de konumlandırılabilir. Bu da 4 farklı geometrik yerleştirmenin daha ortaya çıkmasına sebep olur ve toplam yerleştirme sayısı 8'e çıkar. Ancak, genel kabullenme ağaçtaki sıranın yerleştirmede korunması olacak. Bu 4 olası yerleştirmeye ek olarak, ağaçtaki her bir yaprak, yani A, B ve C bileşenleri, minimum ve maksimum büyüklüklerini de tutmakta olduklarından, tanımladıkları aralıklardaki sınırsız sayıdaki değerlerden herhangi birini genişlik ve yükseklik olarak kabul edebilirler ve düzlemi ayrıklaştırsak dahi, olası mutlak yerleştirmelerin sayısının kolayca binler, hatta milyonlar mertebelerine yükselebileceğini öngörebiliriz. Kaldı ki, bu yalnızca 3 bileşenli bir sistem için yaptığımız çok kaba bir hesaptır. Birazdan bahsedeceğimiz üzere, her bir dalda tercih edilebilecek lokal yerleştirmelerin çeşitliliği ve toplam bileşen sayısı arttıkça, bu ağaç veri yapısı yalnızca doğrusal olarak büyüyecek, ancak üstel olarak büyüyen bir çözüm uzayını içinde barındıracaktır. Şekil-3'deki örnek üzerinden devam ederek aşağıdaki bileşen boyut aralıklarının da yazılımcı tarafından algoritmaya verilmiş olduğunu varsayalım:

$A = (100 - 200), (50 - 150)$
$B = (70 - 140), (100 - 200)$
$C = (50 - 120), (200 - 300)$

Burada görüldüğü üzere, her bileşen için yazılımcının 4 adet nümerik girdi yapması gerekmektedir. Bunlardan ilk ikisi o bileşenin yatay boyut sınırlarını, diğer ikisi ise dikey boyut sınırlarını göstermektedir. Yani, A bileşenin yatay büyüklüğü 100 ila 200 birim arasında sınırlandırılmalı, dikey büyüklüğü ise 50 ila 150 birim arasında olmalıdır. Bu da ayrık bir Öklid uzayında yaklaşık 10000 farklı dikdörtgene karşılık gelmektedir. Benzer şekilde B bileşeni 7000, C bileşeni de yine 7000 farklı dikdörtgen ile gerçeklenebilir. Hemen farkedilecektir ki, A, B ve C bileşenleri için tanımladığımız bu onbinlerce dikdörtgenden yalnızca bir kısmı görsel olarak tatmin edici yerleştirmeleri mümkün kılacaktır. Algoritmanın gerçekleştirdiği de tüm bu olasılıklar içerisinde en "doğru" olasılığı seçmektir.

Ayrıca geliştiricilerin işini kolaylaştırmak açısından boyultandırmaları kademe kademe arttıran öntanımlı değerler girilmiş olup bu değerlerin kartezyen çarpımlarından bir boyutlandırma sistemi geliştirilmiştir. 20 ila 50 pixel arası başlayan boyut aralıkları 890 ila 2000 pixel arası boyutlara kadar ilerlemektedir. Aralıkların düzeltilmesi ve ince ayar yapılması geliştiriciden geliştiriciye değişeceği için istenildiği gibi oynanabilir olarak tasarlanmıştır. Fakat öntanımlı değerler ile oldukça mantıklı sonuçlar elde edilmiştir. Ve yine geliştiricinin işini kolaylaştırmak için "klasik" sayılabilecek form elementleri varsayılan fonksiyonlar olarak tanımlanmıştır. Örneğin geliştirici isim girme alanı yaratmak istese çağıracağı tek fonksiyon ile hem bir etiket alanı hem de yazı girme alanı otomatik olarak eklenip yan-yana veya alt-alta sıralanabilir bir şekilde yaratılacaktır. Elbette bu "klasik" elementlerin dışında geliştiricilerin kendi yazdıkları özel elementleri de desteklemekteyiz. En nihayetinde algoritmamız için bu elementin önemli olan bilgileri; kaplayacağı boyut aralığı ve diğer elementlerle olan ilişkileri. Bu bilgiler dışında hiçbir ekstra bilgiye ihtiyaç duymadan verilen elementleri de bozmadan çalışmaktadır.

3.2. Yerleştirme kriterleri

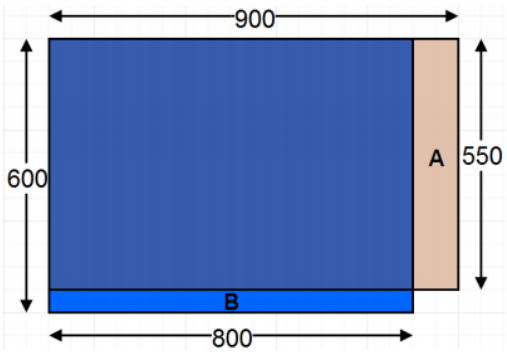
Yukarıda tanımladığımız veri yapısı, herhangi bir kriter gözetmeden tüm olası yerleştirmeleri içinde barındırmaktadır. Ancak girdiler arasında tanımlanması gereken bazı mutlak kriterler söz konusudur. Bunlardan en temel olanı, yerleştirmenin yapılacağı ekranın çözünürlüğüdür. Zira yapılacak yerleştirmenin genişliğinin ekran

genişliğinden, yüksekliğinin ekran yüksekliğinden fazla olması durumlarında yerleştirme ekrana sığmayacaktır. Burada iki durum söz konusudur. İlk durum yerleştirmelerden en az bir tanesinin aralık değerlerinin ekran çözünürlüğü ile örtüşmesidir. Bu durumda, örtüşmeyen yerleştirmeler elenerek örtüşen yerleştirmeler üzerinden işlem yapılmaktadır. Burada elbette tek tek bütün yerleştirmeleri hesaplayıp ekrana sığıp sığmadığına bakmak son derece yavaş ve gerçek zamanlı bir uygulamanın içinde kullanılamayacak kadar hantaldır. Bunun yerine, ağacı yapraklarından köküne doğru tarayarak yapraklardaki boyut aralığı bilgilerini köke taşıyoruz ve kök düğüm tüm yerleşimi sembolize ettiğinden, kök düğümde ortaya çıkacak olan aralık değerleri direkt olarak ekran çözünürlüğü ile kıyaslanabilir olduğu için seçim yapmamızda yardımcı oluyor.

Önceki örneğimiz üzerinden gidersek, dört adet muhtemel yerleştirmenin varlığını göstermiştik. Bunlardan birincisi, yani A bileşeninin B bileşeninin üzerinde, C bileşeninin ise bunların sağında konumlandığı yerleşimi düşünelim. Ayrıca estetik kaygılarla A ve B bileşenlerinin aynı yatay büyüklükte olmaları gerektiğini de varsayalım. A ve B bileşenleri üst üste yerleştirildiğinde oluşan A ve B'yi içeren en küçük dikdörtgene X diyelim. X dikdörtgeni, en az 100, en fazla 140 birim yatay büyüklüğünde olabilir. Aksi takdirde, ya A ya da B bileşeninin sınır değerleri ihlal edilmiş olur. Benzer şekilde X dikdörtgeninin alt ve üst dikey büyüklük sınır değerleri 150 ve 350 olarak bulunur. Zira üst üste yerleştirilen bileşenlerin dikey sınır değerlerini toplayarak yeni sınır değerler bulunabilir. Bu durumda X dikdörtgeninin sınır değerleri (100-140),(150-350) olmuş olur. X ile C bileşenleri yanyana yerleştirileceği için benzer şekilde X ile C'yi içine alan en küçük dikdörtgenin ki buna da Y dikdörtgeni diyelim, sınır değerleri de (150-260),(200-300) olarak bulunur. Böylece Şekil-3'te sol üstte gösterilen yerleştirmenin gerçekleşmesi durumunda sınır değerlerinin (150-260),(200-300) olacağını bulmuş oluruz. Diyelim ki, 200x250 çözünürlüğe sahip bir ekranda bu uygulamayı çalıştırmayı istiyoruz. Bu durumda algoritmamız bu yerleştirmenin verilen ekran çözünürlüğü için kullanılabileceğini bulup uygun yerleştirmeleri sıralamaktadır. Ancak ekran çözünürlüğümüz 400x200 olsaydı, o zaman yatay büyüklüğü sağlayamadığımız için, algoritmamız bu yerleştirmeyi muhtemel yerleştirmelerin arasına almayacaktı.

Dikkat edilirse, ağaçtaki her bir dal düğümü alternatif iki yerleştirme sunmaktadır: yatay ve dikey. Bu durumda k adet dal düğümü olan bir ağaçta 2^k adet yerleştirme için bu hesaplamanın yapılması gerekir. 10 bileşenli bir ekranda bu çok sorun olmayabilir. Ancak bileşen sayısı arttıkça bu rakam hızla pratik sınırların üzerine yükselecek ve gerçek zamanlı bir uygulamanın bu kadar yerleştirmeyi tek tek incelemesi mümkün olmayacaktır. Bu durumda algoritmamız yerleştirmeleri kök düğüme ulaşmadan filtreliyor. Örneğin bir dal düğümün olası yatay büyüklük sınırları (800-1200) iken, ekran çözünürlüğünün yatay bileşeni 600 olarak verilmiş ise, bu durumda o dal düğümünün sınır aralığını yukarıya taşımanın bir anlamı kalmayacağı için bu dal düğümün hesaplanması durduruluyor. Bu şekilde pek çok yerleştirme, daha dal düğümleri hesaplanırken eleniyor ve ağaç süratle budanabiliyor. Ayrıca bu süreci hızlandırmak için yapılan başka bir işlem ise hatırlama (memoization) mekanizmasıdır. Hatırlama mekanizması oldukça basit olup, eğer ki aynı çözünürlük için tekrardan uygun ağaç yapısını çıkartmak istiyorsak, ki bu durum tüm ağacımızı dolanırken sıkça karşılaşacağımız bir durum, o zaman hiçbir parametremiz değişmediği için tekrardan hesaplamamız bize sadece performans düşüşü olarak geri dönecektir. Az önceki örnek üzerinden ilerleyecek olursak, Y dikdörtgeninin içindeki elemanların olası yükseklik ve genişliğini hesaplamak istediğimizde X dikdörtgeninin genişlik ve yükseklik hesabını zaten yaptığımızı görebiliriz ve bu hesap çözünürlük veya ağaç değişmediği için o anlık sabit kalacaktır. Bu sebeple bizim tekrardan bir hesap yapmamıza gerek kalmamaktadır. Fakat şayet başka bir çözünürlük için bu hesabı yapıyor olsaydık, bu kayıtta tutulan bilgilerin temizlenmesi gerekirdi.

En başta bahsettiğimiz ikinci durumda ise, hesaplanan yerleştirmelerin hiçbirisi ekrana sığmıyordur. Bu durumda olası iki hareket planımız söz konusudur. Birincisi kullanıcıya uygulanabilir bir yerleştirme bulunmadığının bildirilmesidir ki bu pek çok durumda tercih edilen bir yol olmayacaktır. İkincisinde, hedef çözünürlüğe en yakın olan yerleştirme önerilip, bu yerleştirme arayüzün kaydırma çubuklarıyla desteklenmesiyle sunulabilir. Örneğin, ekran çözünürlüğünün 800x600 pixel olduğu bir durumda, olası iki kök düğüm yerleştirmesinden birincisinin (900-1200),(480-560), ikincisinin (750-850),(700-900) boyut aralıklarında olduğunu farz edelim. Yani birinci yerleştirmeyi göz önünde bulundurursak, bu bize minimum 900, maksimum 1200 birim yatay büyüklüğe sahip ve yine minimum 480, maksimum 560 birim dikey büyüklüğe sahip yerleştirmeler yapma imkanı sunmaktadır. Ancak görüldüğü üzere, yatay büyüklüğün minimum sınırı olan 900 birim dahi istenen 800 birimin üzerinde kalmaktadır. Dolayısıyla bu yerleştirmenin tanımladığı tüm gerçeklemeler istenilen değerin dışında kalmaktadır. Aynı şey ikinci yerleştirmede de dikey boyutda söz konusu olmaktadır. Dolayısıyla algoritma iki yerleştirmeden estetik olarak daha yüksek puan alanını seçerek, kaydırma çubuklarıyla arayüzü oluşturmaktadır. Daha uygun olanın belirlenmesinde, sınır değerler üzerinden hesaplanan hata miktarları ve arayüzün estetik puanı karşılaştırılır. Örneğin, birinci yerleştirmede yatay büyüklük alt sınır değeri olan 900 birim sabitlenerek yerleştirme yapıldığında yerleştirme dikey büyüklüğünün 550 birim olarak hesaplandığını varsayalım. Bu durumda 900x550 büyüklüğünde bir yerleştirme gerçeklemesi elde etmiş oluruz, bunun da hata miktarı aşağıdaki şekilde gösterildiği gibi hesaplanır:



Şekil 4. Birinci yerleştirme hata miktarı hesaplaması: Yerleştirme bej, ekran çözünürlüğü mavi olarak verilmiştir.

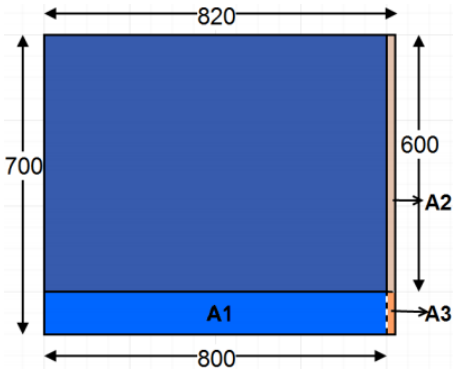
Burada A ve B ile işaretlenen bölgeler hata miktarlarını göstermektedir. A bölgesi, yerleştirmenin ekrandan taşan kısmını gösterirken, B bölgesi ise ekranın yerleştirme tarafından kullanılmayan kısımlarını göstermektedir. İdeal bir yerleştirmede A+B toplamı 0 olmalıdır. Bunun mümkün olmadığı durumlarda A+B toplamını minimize edecek yerleştirme en uygun yerleştirme kabul edilir. Bu yerleştirme örneği için hesaplanacak olan değer

$A = 550 \times (900 - 800) = 55000$
$B = 800 \times (600 - 550) = 40000$
$A + B = 55000 + 40000 = 95000$

olarak bulunur. İkinci yerleştirmede yükseklik aralığı ekran yüksekliğiyle örtüşmediği görülür. Bu sebeple yükseklik alt sınırı olan 700 değerini sabitleyerek yapılan yerleştirmede 820 pixel genişlik ortaya çıktığını varsayalım. Bu durumda gerçeklenen yerleştirme 820x700 pixel olacaktır. Benzer şekilde hata miktarı hesaplandığında:

$A1 = 800 \times (700 - 600) = 80000$
$A2 = 600 \times (820 - 800) = 12000$
$A3 = (820 - 800) \times (700 - 600) = 2000$
$A1 + A2 + A3 = 80000 + 12000 + 2000 = 94000$

olarak bulunur.



Şekil 5. İkinci yerleştirme hata miktarı hesaplaması: A1, A2 ve A3'ün toplamı hata miktarını vermektedir.

Burada A1, A2, A3 değerleri yerleştirmenin ekrana sığmadığı bölge 3 dikdörtgene bölünerek elde edilmektedir. Toplam hata payı ilk yerleştirmede 95000 pixel, ikinci yerleştirmede 94000 pixel olarak hesaplandığı için, ikinci yerleştirme daha uygun bir çözüm olarak görülmektedir.

Algoritma ayrıca, bileşen boyutlarına mutlak değerler atayarak ekran çözünürlüğüne karşılık gelen yerleşim seçeneklerini sıralar. Buradaki zorluk ise, her bir yerleştirmenin aslında devamlı uzayda sonsuz adet çözüme sahip olmasıdır. Ayrıklaştırma yöntemiyle dahi, yüzlerce çözümün tek tek elden geçirilmesi gerekir. Bu yüzden kaba kuvvet algoritmalarının kullanılması çok zor görülmektedir. Bunun yerine, uyguladığımız yöntemin ana hatları şu şekildedir:

Tüm olası yerleştirmelerin yatay ve dikey sınırları hesaplandıktan sonra bu sınırları kullanarak ve ağacı yukarıdan aşağıya tarayarak, tüm düğümlerin yatay ve dikey sınır değerleri bu değerler arasına sınırlanır. Bir başka deyişle, tüm bileşenler, yukarıdan atanacak sınır değerine uymak zorunda kalır.

Örneğin, A ve B bileşenlerinin yatay olarak yerleştirildiğini, ve yatay sınır değerlerinin A için (50-150), B için (100-200) olarak verildiğini kabul edelim. Dolayısıyla A ve B'nin birleşiminden oluşan düğümün yatay sınır değeri (150-

350) olur. Ancak bu düğümün yatay büyüklüğü 220 olan bir dikdörtgene sığdırılmak istendiğini düşünelim. Bu durumda A'nın yatay büyüklüğünün 150 olması mümkün değildir. Zira buna karşılık gelen B yatay değeri 70 olacak ve bu da B'nin sınırları dışında kalacaktır. Bu yüzden hem A, hem de B'nin yatay sınırları traşlanarak A için (50-120) ve B için (100-170) değerleri bulunur. Bu yeni değerler içinde A'nın alacağı her değere karşılık B'nin alabileceği bir değer mevcuttur.

Bu aşamada, ağaç yukarıdan aşağıya doğru taranarak ekran çözünürlüğü kısıtları tüm dal ve yapraklara uygulanır. Bir başka deyişle, belirlenen aralıklar, ekranın yatay ve dikey büyüklüğünü birebir yansıtacak şekilde tek bir değere indirgenir. Bu indirgeme işlemi iki türlü olabilir: Eğer bileşenleri dikey yerleştirilmiş bir kümenin yatay değerlerini sabitlemek istiyorsak, kümenin her bir bileşenin yatay büyüklüğünü kümenin yatay büyüklüğüne eşitleriz. Eğer bileşenleri yatay yerleştirilmiş bir kümenin yatay büyüklüğünü sabitlemek istiyorsak, o zaman kümenin yatay büyüklüğünü bileşenler arasında dağıtırız. Bu dağıtma işleminin hızlı ve estetik olarak yapılması algoritmanın en kritik fonksiyonlarından bir tanesidir. Bunun için, Borning'in kullandığı simplex benzeri optimizasyon rutinlerinden faydalanılabileceği gibi, açgözlü ve sezgisel metotlar yardımıyla daha hızlı çözümler de üretilebilirdi ve biz algoritmamızda açgözlü ve sezgisel metotları tercih ettik ve özel bir kontrol mekanizması yarattık.

3.3. Algoritma

Bu mekanizmadan bahsetmeden önce ağacın nasıl yaratıldığını göstermek gerekirse, ağaçtaki yapraklara Yaprak, dal düğümlere de Dal dersek, Şekil-3'teki örnek ağacı şu şekilde yaratabiliriz. elemanYarat metodu ile yaprak veya dal düğüm yaratıp birbirleri arasındaki ilişkileri tanımlanır. Örnek kodda görüldüğü gibi A ve B yaprakları yaratıldıktan sonra X dalına bağlanır ve burada sıraya önem verilir. Bileşen boyutu bilgisi tasarım sürecini hızlandırmak için öntanımlı değerler arasından verilir, ancak istenirse bu bilgi daha hassas olarak da fonksiyona verilebilir.

```
1  Yaprak A = elemanYarat("A", new JLabel(),
2      BilesenBoyutu.KUCUK_KUCUK);
3  Yaprak B = elemanYarat("B", new JLabel(),
4      BilesenBoyutu.KUCUK_KUCUK);
5  Dal X = elemanYarat("X", A, B);
6  Yaprak A = elemanYarat("C", new JLabel(),
7      BilesenBoyutu.KUCUK_ORTA);
8  Dal M = elemanYarat("M", X, C);
9  return M;
```

Şekil 6. Üç elemanlı örnek ağaç yapısının yaratılışı

Şekil-6 üzerinde görüldüğü gibi X bir dal olup kullanıcılara gösterilmeyen bir taşıyıcıdır. İçinde A ve B'nin ilişkisel dizilimi ve toplam kaplayacakları minimum ve maksimum alan gibi bilgiler tutulur. Dal ve yapraklarda Genişlik-Yükseklik-Aralığı adlı bir yapı yer almaktadır ve bu yapı, minimum ve maksimum, genişlik ve yükseklik değerlerini tutar. Ayrıca bunların dışında, bu yapıda alt kırılımların Genişlik-Yükseklik-Aralığı bilgileri ve yatay düzlemi mi, dikey düzlemi mi, yoksa ağacın bir uç elemanını mı temsil ettiğini tutan bir değerimiz de vardır.

Algorithm 1: GETRANGES Gerçeklenebilir yerleştirmeleri tespit etme algoritması

Input: Formun ağaç bilgisini tutan veri yapısı
Output: Formu temsil eden ağaca ait WidthHeightRange objesi

```
1  if memo ≠ ∅ then return memo
2  movingRanges ← ∅ WidthHeighRange'den oluşan
3  tempRanges ← ∅ WidthHeighRange'den oluşan
   // Yatay olasılıklar
4  foreach child in children do
5      compareList ← child.GET-RANGES
6      if movingRanges = ∅ then
7          compareList'teki tüm elemanları
             movingRanges'ın içine ekle
8      else
9          foreach whr in movingRanges do
10             foreach newWhr in compareList do
11                 if newWhr, whr ile yatay
                     yerleştirilebilir ise then
12                     newWhr'yi tempRanges'a ekle
13         movingRanges ← ∅
14         tempRanges'daki tüm elemanları
             movingRanges'a ekle
   // Dikey olasılıklar
15 foreach child in children do
16     compareList ← child.GET-RANGES
17     if movingRanges = ∅ then
18         compareList'teki tüm elemanları
             movingRanges'ın içine ekle
19     else
20         foreach whr in movingRanges do
21             foreach newWhr in compareList do
22                 if newWhr, whr ile dikey
                     yerleştirilebilir ise then
23                     newWhr'yi tempRanges'a ekle
24         movingRanges ← ∅
25         tempRanges'daki tüm elemanları
             movingRanges'a ekle
26 memo ← movingRanges
27 return movingRanges
```

Şekil 7. Gerçeklenebilir yerleştirmeleri tespit etme algoritması

Ağaçtaki dal ve yaprakların ortak özellikleri olduğu gibi farklılıkları da vardır. Her elemanda ortak olarak atanan X ve Y koordinatları tutulur. Bunlar sayfada nereye konumlanacaklarını temsil eder. Ayrıca atanan genişlik ve yükseklik değerleri de vardır ve bunlar da boyut değerleridir. Dallarda ise altındaki kırılımın tamamının Genişlik-Yükseklik-Aralığı'nı öğrenebileceği geri dönüşümlü bir algoritma bulunmaktadır. Bu algoritma verilen daldaki gerçekleştirilebilir yerleştirmeleri tespit etmeyi sağlar.

Şekil-7'de görüleceği üzere bu kendini çağıran metodun optimizasyonu için çeşitli işlemler uygulanmıştır. En başında ise, her seferinde baştan yapılan işlemlerin önüne geçilmiştir. Alt kırılımlarda hesaplanan birimler satır 26 sayesinde kaydedilir ve satır 1 sayesinde tekrar kullanılır. Böylelikle aynı çözünürlükte tekrar eden hesaplamaların önüne geçilmiş olur.

Şayet hesaplama daha önce yapılmamışsa şu şekilde yapılmaktadır: Öncelikle alt kırılımlarına bakılacak olan kök birimin çocuklarına bakılır. Her bir çocuğun Genişlik-Yükseklik-Aralığı bulunur ve bu değerler bir karşılaştırma listesine alınır. Eğer "movingRanges" listesi boş ise hepsi bu listeye eklenir ve bu liste bizim olası yatay dizilimlerimizin birkaçını taşımaya başlar. Eğer ki bu listemiz boş değilse, karşılaştırma listemiz ile olası yatay yerleştirmelerimiz çaprazlanır ve fizibilite kontrolleri yapılır. Bu kontrollerden geçen Genişlik-Yükseklik-Aralığı nesneleri ise satır 14 ve 25'te yatay veya dikey olmasına göre "movingRanges" listesine eklenir ve böylelikle

elimizde kök birimin tüm olası yatay veya dikey dizilimleri bulunur. Fizibilite kontrollerini geçenleri geçici bir listeye aldıktan sonra elimizde tüm yatay ve dikey dizilimler bulunmuş olur.

Algorithm 2: LAYOUT Eşit Dağıtım algoritması

Input: Formu temsil eden ağaca ait WidthHeightRange objesi
Output: Formdaki her bir öğenin kesin koordinatları, genişlik ve yüksekliği

```
1  feasible ← false
2  distribution ← ∅ children sayısı kadar
3  remaining ← yataya veya dikeye göre kalan değer
4  foreach child in children do
5      distribution[current] ← bu çocuğun dağıtılmış
         minimum değerleri
6      remaining −= distribution[current]
7  while remaining > 0 do
8      if remaining/∑remainingChild < minimum
         then
9          remaining/∑remainingChild değerini
             dağıt
10     else
11         minimum değerini dağıt
12 offset ← 0
13 foreach child in children do
14     if orientation = Horizontal then
15         if ¬child.LAYOUT (x+offset, y,
             distribution[current], height) then
16             return false
17     else
18         if ¬child.LAYOUT (x, y+offset, width,
             distribution[current]) then
19             return false
20     offset += distribution[current]
21 return true
```

Şekil 8. Eşit Dağıtım algoritması

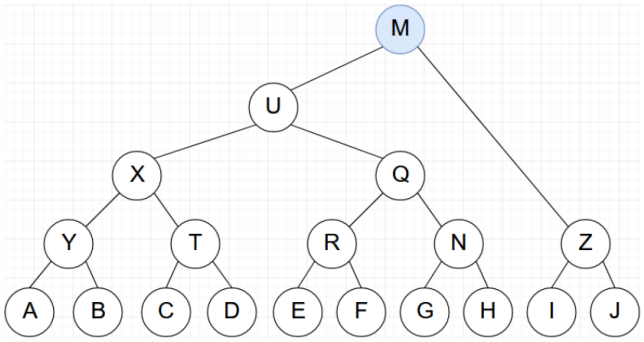
Hazırlanan bu ağaç yapısının çalıştığını göstermek için Java dilinde bir prototipleme çalışması yapıldı ve bu algoritma görsellere döküldü. Pencere her tekrardan boyutlandırıldığında çalışan bu algoritma için bir de dağıtım algoritması yazıldı. Örnek bir dağıtım stratejisi olarak da eşit dağıtma seçildi.

Eşit dağıtım algoritmasının çalışma mantığı, Şekil-8'deki satır 5'te olduğu gibi öncelikle minimumları dağıtıp ardından kalanları çocuklar arasında eşit dağıtmasıdır. Kalan çocuk sayısı belirlenirken, çocukların minimum ve maksimum limitler arasında olup olmadığı kontrol edilir, aksi takdirde çocuğa fazladan eklenen değer o dalın fizibilitesini bozar ve dolayısıyla tüm ağacın fizibilitesi bozulacağı için o dizilim geçersiz hale gelir. Şekil-8 satır 7 ile 11 arasında çocuklar arasında alınabilecek en düşük değerin altında bir rakam kalmışsa o kalan değer dağıtılacak çocuk sayısına bölünerek her uygun çocuğa eklenir. Diğer türlü çocuklar arasındaki en düşük değer dağıtılır. Şekil-8 satır 13'den sonrasının anlattığı ise, son kalan değerler de paylaştırıldıktan sonra her çocuk için dağıtım fonksiyonu geri dönüşümlü olarak çağırılır ve bu sayede ağaç kökten yapraklara doğru işlenerek gider. Her dal kendi içinde dağıtımlarını hesaplayarak en sonunda yapraklara kadar ilerler ve her bir yaprağın nerede konumlanacağı ve genişlik ve yükseklik değerleri kesin bir şekilde belirlenmiş olur.

Daha sonra çizim yapılırken olası dizilimlerden seçilen dizilimin elemanları tek tek gezilir ve tüm yaprak birimlerin atanan koordinatları ve boyutlarına göre sayfaya yerleştirilmeye başlanır. Burada seçim geliştiriciye de bırakılabilir ya da otomatik olarak öklid uzaklığına göre en yakın dizilim uygulanabilir; prototip uygulama öklid uzaklığına bakarak en uygun dizilimi gösterir.

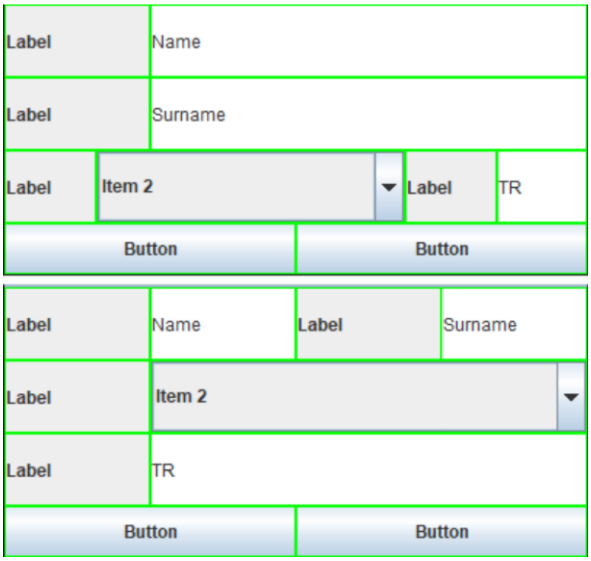
4. Araştırma bulguları

Elde edilen birkaç örnek sonuçta, bir ağacın iki farklı çözünürlükte, iki de aynı çözünürlükte fakat farklı dizilimde olan sonuçları ele alındı.



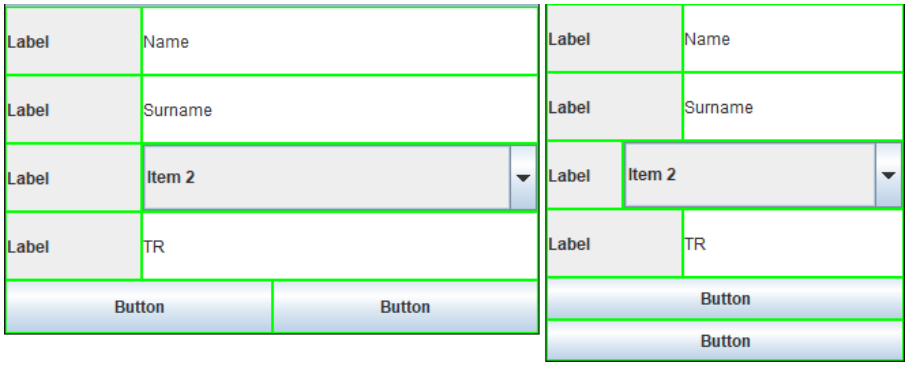
Şekil 9. Ağacın tasarımcı tarafından ilişkisel olarak tanımlandığı görsel

Şekil-9 ağacın yapısını temsil eder. M isimli kök ağacın en tepesini gösterirken A'dan J'ye tüm birimler yaprakları gösterir. Arada kalan tüm birimler ise dal olarak nitelendirilir.



Şekil 10. Aynı ağacın, aynı çözünürlükte iki farklı dizilimi

Şekil-10'da örnek ağacın aynı çözünürlükte önerdiği iki farklı dizilimi görüyoruz. Yaprak birimlerin isimlendirmesini sol üstten sağa ve aşağı doğru yapacak olursak, üstteki dizilimde, A, B ve C, D elemanları kendi içlerinde yatay birbirleri arasında dikey dizilim gösterirken E, F ve G, H elemanları hem kendi içlerinde hem de birbirleri arasında yatay dizilim göstermektedir. Alttaki örnekte ise tam tersi bir durum önerilmiştir. Aynı boşlukları doldurabilecek bu dizilimde ise A, B ve C, D elemanları hem kendi içlerinde hem de birbirleri arasında yatay dizilim gösterirken E, F ve G, H elemanları kendi içlerinde yatay birbirleri arasında dikey dizilim göstermektedir.



Şekil 11. Aynı ağacın, farklı çözünürlükte iki farklı dizilimi

Farklı çözünürlüklerde çeşitli önerilere bakacak olursak birbirine yakın iki çözüm üretilen Şekil-11'e bakılabilir. Burada her bir elementin minimum maksimum değerleri zorlandığında nasıl bir davranış gösterdiği incelenebilir. Örneğin üçüncü satırdaki çoktan seçmeli kutunun Genişlik-Yükseklik-Aralığı diğer yazı alanlarından farklıdır. Bu yüzden formu küçülttüğümüzde diğer elemanlardan farklı davranır. Diğer bir bakış açısında ise soldaki etiketlerin hepsi aynı Genişlik-Yükseklik-Aralığı'na sahiptir. Fakat bu aralık oldukça esnek olduğu için çoktan seçmeli kutuyu tolere edecek kadar küçülebilir. Böylelikle daha düşük bir çözünürlükte bile formun bütünlüğünü koruyabilir. Aynı şekilde butonlar da yan yana sığamayacağını anlayınca alt alta dizilimin daha mantıklı olduğunu görmüştür ve dizilimini değiştirmiştir. Genişlik-Yükseklik-Aralığı içerisinde uygun yaratılan bu iki dizilimde kolaylıkla bu küçük detaylar göze çarpmaktadır.

Bu örneklerde olduğu gibi daha bir sürü farklı örnekler üretilebilir. Fakat bu örneklerin bize gösterdiği ise bu algoritmanın her olasılığı düşünmesiyle oldukça değişik yeni bakış açıları getirebildiğidir. Bir tasarımcının bile aklına gelemeyecek önerilerde bulunabilen bu algoritma çok daha güçlü sonuçlar çıkarabilecek kapasitede olup estetik kaygı güderek geliştirilmesi daha verimli sonuçlar elde edilmesini sağlayacaktır.

Geliştirdiğimiz yöntemin çalışma sürelerini incelemek için algoritmalarımızı 19, 33, ve 84 bileşenli 3 farklı arayüz tasarımı üzerinde test ettik. Elde ettiğimiz sonuçlar algoritmaların bu farklı arayüz tasarımlarını ne kadar sürede ekrana çizilebilir geometrik çıktıya dönüştürebildiğini göstermektedir.

Tablo 1. Şekil-7'ye ait çalışma süreleri

Bileşen Sayısı	Hatırlama Olmadan	Hatırlama Varken
19	584 nanosaniye	292 nanosaniye
33	395982 nanosaniye	585 nanosaniye
84	84421869 nanosaniye	877 nanosaniye

Şekil-7'deki ağaç yapısını çıkartma algoritmasını ele alınırsa, uygulama başlatıldığında ağacımızın yapısını çıkartırken çağırılan bu fonksiyonu yapılan budama yöntemleriyle beraber hatırlama (memoization) kullanılan ve kullanılmayan hallerini hesaplarken harcanan zamanı Tablo-1 de görebilirsiniz. Ağaç ilk kez yaratılırken harcanan süre ile önceden hesaplanmış ve sadece değer geri dönen dallara harcanan süre arasında oldukça büyük bir fark gözlemlenmiştir. Burada olası seçenekler kümemizin eleman sayısı 19 bileşende 2^{19}=524288 iken 33 bileşende 2^{33}=8589934592 farklı kombinasyona ulaşıyor. Elde edilen sayılarla budama yöntemimizin geliştirmeye açık olmasıyla beraber ne kadar etkili olduğunu görebiliriz.

Tablo 2. Şekil-8'e ait gerçeklenebilir ve gerçeklenemez koşullarda yerleştirme süreleri

Bileşen Sayısı	Gerçeklenemez	Gerçeklenebilir
19	292 nanosaniye	31561 nanosaniye
33	1169 nanosaniye	41206 nanosaniye
84	3506 nanosaniye	238466 nanosaniye

Hazırlanan ağacın kullanımındaki sürelere bakacak olursak, Şekil-8 özelinde işlenen ağaca harcanan zaman gerçeklenebilir dizilim ile gerçeklenemez dizilim arasında farklılık göstermektedir. Sebebi ise yine gerçeklenemez dizilimleri ağacı dolaşırken elememizdir. Örneğin Şekil-9 özelinde çizilen ağaçta Y dalında gerçeklenemez olduğu kanaatine varıldığı anda; yani daha Y dalında bu dizilimin ekrana sığmayacağı belirlendiğinde ağacın kalanına bakılmaksızın bir sonraki olasılığa geçiliyor ve böylelikle ağaç oluşturulurken elenenler kümesinden kalanlar dizilim sırasında bir daha elenerek en hızlı çözüme ulaşılmaya çalışılıyor. 84 bileşene kadar çıkıldığında dahi gerçek zamanda çalışmanın ötesinde, algoritma 1 milisaniyenin altında sonuçlanabiliyor. Bu hesaplamalar Java dilinin ve arayüzdeki değişimlerin uygulanmasını kapsamamaktadır ve sadece algoritmanın çalışma süresini tutmaktadır. Bu da geliştirdiğimiz algoritmanın rahatlıkla yüksek karmaşıklığa sahip arayüzlerin gerçek zamanda otomatik olarak yerleştirilmesi amacıyla kullanılabileceğini göstermektedir.

4. Sonuç ve tartışma

Bu makalede uygulama arayüzü geliştirme süreçlerinde kullanılabilecek, tasarımcı ve programcıların işlerini kolaylaştıracak, uygulama geliştirme süreçlerini kısaltacak, uygulamaların ürüne dönüştükten sonraki bakım ve güncelleme maliyetlerini azaltacak bir arayüz geliştirme algoritmasını sunduk. Bu algoritma, gerçek zamanlı olarak son derece karmaşık arayüz tasarımlarını belirlenen ekran çözünürlüğüne en uygun şekilde yerleştirmektedir. Bunu yaparken belirlenen minimum kriterlere bağlı kalarak, öncelikle olası tüm yerleştirmeleri hesaplamakta, ardından bu yerleştirmelerin içinden istenilen bir tanesini seçerek ekran çözünürlüğüne tam oturacak şekilde arayüz içindeki bileşenlerin geometrik değerlerini hesaplamaktadır. Tüm bu işlemler saniyenin

çok altında bir sürede yapılabilmekte olup, gerektiğinde masaüstünde pencerenin yeniden boyutlandırması esnasında dahi gerçek zamanlı çalıştırılabilecek kadar hızlıdır.

Önerdiğimiz bu algoritma gerektiğinde masaüstü uygulamaları, web uygulamaları, web sayfaları ve mobil uygulamalar için uyarlanabilecek şekilde tasarlanmış olup, herhangi bir programlama dili veya kütüphanesinden bağımsız olarak çalışabilmektedir. Bu özelliğiyle, farklı platformlarda kullanılabilmesi mümkündür.

İleride bu algoritmanın daha iyi hale getirilmesi için kullanıcıya daha güzel gelecek, estetik kriterlere dikkat eden ve yerleştirmeyi evrensel estetik kabuller üzerinden yapabilen bir algoritmanın çalışılması mümkündür. Ancak evrensel estetik kabullerin tespiti son derece zor bir konu olduğundan, bu çalışma kapsamında buna yer verilmemiştir.

Çıkar Çatışması (Conflict of Interest)

Yazar tarafından herhangi bir çıkar çatışması beyan edilmemiştir. No conflict of interest was declared by the author.

Kaynaklar (References)

Badros, G. J., Borning, A., & Stuckey, P. J. 2001. The Cassowary linear arithmetic constraint solving algorithm. ACM Transactions on Computer-Human Interaction (TOCHI), 8(4), 267-306.

Borning, A., Lin, R., & Marriott, K. 1997, November. Constraints for the web. In Proceedings of the fifth ACM international conference on Multimedia (s. 173-182).

Borning, A., Marriott, K., Stuckey, P., & Xiao, Y. 1997, October. Solving linear arithmetic constraints for user interface applications. In Proceedings of the 10th annual ACM symposium on User interface software and technology (s. 87-96).

Buanga, P. M. 2011. Automated evalution of graphical user interface metrics.

Gajos, K., & Weld, D. S. 2004, January. SUPPLE: automatically generating user interfaces. In Proceedings of the 9th international conference on Intelligent user interfaces (s. 93-100).

Hosobe, H. 2000, September. A scalable linear constraint solver for user interface construction. In International Conference on Principles and Practice of Constraint Programming (s. 218-233). Springer, Berlin, Heidelberg.

Hosobe, H. 2011, November. A simplex-based scalable linear constraint solver for user interface applications. In 2011 IEEE 23rd International Conference on Tools with Artificial Intelligence (s. 793-798). IEEE.

Jacobs, C., Li, W., Schrier, E., Bargeron, D., & Salesin, D. 2003. Adaptive grid-based document layout. ACM transactions on graphics (TOG), 22(3), 838-847.

Jamil, N. 2014. Constraint solvers for user interface layout. arXiv preprint arXiv:1401.1031.

Jamil, N., Müller, J., Naeem, M. A., Lutteroth, C., & Weber, G. 2016. Extending linear relaxation for non-square matrices and soft constraints. Journal of Computational and Applied Mathematics, 308, 346-360.

Jamil, N., Needell, D., Muller, J., Lutteroth, C., & Weber, G. 2013, Kasım. Kaczmarz algorithm with soft constraints for user interface layout. In 2013 IEEE 25th International Conference on Tools with Artificial Intelligence (s. 818-824). IEEE.

Jiang, Y., Du, R., Lutteroth, C., & Stuerzlinger, W. 2019, May. ORC layout: Adaptive GUI layout with OR-constraints. In Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (s. 1-12).

Lutteroth, C., Strandh, R., & Weber, G. 2008. Domain specific high-level constraints for user interface layout. Constraints, 13(3), 307-342.

Nielsen, J. 1994, April. Usability inspection methods. In Conference companion on Human factors in computing systems (s. 413-414).

Pajusalu, M., Torres, R., & Lamas, D. 2012. The Evaluation of User Interface Aesthetics. Masters, Tallinn University.

Roscher, D., Lehmann, G., Schwartze, V., Blumendorf, M., & Albayrak, S. 2011. Dynamic distribution and layouting of model-based user interfaces in smart environments. In Model-Driven Development of Advanced User Interfaces (s. 171-197). Springer, Berlin, Heidelberg.

Roudaki, A., Kong, J., & Yu, N. 2015. A classification of web browsing on mobile devices. Journal of Visual Languages & Computing, 26, 82-98.

Sottet, J. S., Calvary, G., & Favre, J. M. 2006. Models at runtime for sustaining user interface plasticity. In Models@ run. time workshop (in conjunction with MoDELS/UML 2006 conference).

Zeidler, C., Lutteroth, C., & Weber, G. 2012, Temmuz. Constraint solving for beautiful user interfaces: how solving strategies support layout aesthetics. In Proceedings of the 13th International Conference of the NZ Chapter of the ACM's Special Interest Group on Human-Computer Interaction (s. 72-79).

Zeidler, C., Lutteroth, C., Sturzlinger, W., & Weber, G. 2013, Ekim. The auckland layout editor: an improved GUI layout specification process. In Proceedings of the 26th annual ACM symposium on User interface software and technology (s. 343-352).

Zeidler, C., Weber, G., Stuerzlinger, W., & Lutteroth, C. 2017, Eylül. Automatic generation of user interface layouts for alternative screen orientations. In IFIP Conference on Human-Computer Interaction (s. 13-35). Springer, Cham.

Zukowski, J. 1997. Java AWT reference. O'Reilly Media.