GAZİ UNIVERSITY

COMPUTER ENGINEERING

Gökay Dindar - 181180024

Genetik Algoritmalar Final Raporu
Genetik Algoritma Kullanarak Sudoku Çözümü

A Retrievable Genetic Algorithm for Efficient Solving of Sudoku Puzzles
Seyed Mehran Kazemi, Bahare Fatemi

Doç. Dr. Ümit ATİLA

İÇİNDEKİLER

1.GİRİŞ

1.1. Problemin Kısaca Tanıtılması

Klasik Sudoku, 3×3 alt matrislere bölünmüş 9×9 matristen oluşur ve birden dokuza kadar olan sayıları her sütun ve satırda birer kez kullanarak kutuların içine yerleştirmesi gerekir. Her satırda, 1 ila 9 arasındaki her sayı sadece bir kez kullanılmalıdır. Yani, her sayı bir satırda yalnızca bir defa bulunabilir. Her sütunda, 1 ila 9 arasındaki her sayı sadece bir kez kullanılmalıdır. 9x9'luk kare, 3x3'lük alt karelere bölünmüştür. Her bir 3x3 alt kare içinde, 1 ila 9 arasındaki her sayı sadece bir kez kullanılmalıdır. Sudoku için önceden belli sayıda sayı kurallara uygun biçimde verilmektedir.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Şekil 1.0 Sudoku Örneği

1.2. Makalenin Bahsettiği Geleneksel Yaklaşım Yöntemleri

Sudoku problemlerini polinom zamanda çözmemiz için P=NP olmalı ancak şuan için polinom zamanda çözen bir algoritma bulunmamıştır. Önceki araştırmacılar sudoku bulmacalarını çözmek için heuristik algoritmalar önermeye veya Genetik Algoritma (GA) gibi mevcut yerel arama yöntemlerini kullanmaya çalışmışlardır.

1.3. Geçmiş Çalışmalar Ve Literatür Taraması

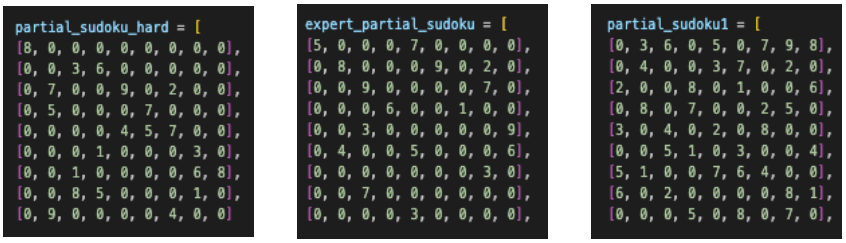
Sudoku bulmacalarını çözmek için literatürde önerilen bir yöntem Pencil and pen algoritmasıdır. Bu algoritma, bir çözüm bulunana kadar bir ağaç üzerinde geri izleme yaparak çalışan bir ağaç tabanlı arama algoritmasıdır. Literatürdeki diğer bir yöntem ise meta-sezgisel bir teknik kullanmaktadır. Bu, simulasyonla ısıl işlem kullanarak mantıksal olarak çözülebilen bulmaca durumlarını etkili bir şekilde çözebilen stokastik bir arama tabanlı algoritmadır. Stokastik optimizasyon, bu probleme uygulamaya çalışan araştırmacıların başvurduğu bir başka yaklaşımdır. Bu yaklaşımda, Sudoku bulmacası stokastik yerel arama teknikleri kullanılarak çözülür. Kültürel Genetik Algoritma, İtici Parçacık Sürü Optimizasyonu, Kuantum Simülasyonlu Isıl İşlem ve Genetik Algoritma ile Simülasyonlu Isıl İşlemi Birleştiren Hibrit Yöntem, stokastik optimizasyon yöntemlerini kullanan bazı yöntemlerden sadece birkaçıdır.

1.4. Makalenin Genetik Algoritma Yaklaşımı

Genetik algoritmalar kullanılarak Sudoku çözümü sağlamak için bir yöntem sunulmuş ve bu yöntem, literatürdeki başka bir genetik algoritma temelli yöntemle karşılaştırılarak çözüm süreleri zorluk seviyelerine göre değerlendirilmiştir. Bir Sudoku bulmacasını 9×9 matris olarak görebiliriz, burada her bir satır, sütun ve her 3×3 alt karede 1'den 9'a kadar olan tüm sayılar yalnızca bir kez bulunmalıdır.GA da Sudoku bulmacasını temsil etmek için, her kromozomu 9×9 boyutunda iki boyutlu bir tamsayı dizisi olarak temsil ediyoruz ve bunlara ek olarak sudoku çözümümüz aşağıdaki koşulları sağlamalıdır;

İlk verilen sudoku parçasını kodumuzda *partial sudoku* olarak adlandırdık.

1) Sudoku önceden belirlenmiş sayılar, kromozomdaki ilgili pozisyonlarında sabit olmalı ve değiştirilmemelidir bilinmeyen sayılar 0 olarak işaretlenmiştir.



Şekil 1.1 Partial Biçimde Algoritmaya Verilmiş Sudokular

2) Kromozomun her bir satırı yalnızca 1 ile 9 arasındaki sayıları bir kez içermelidir,

```
empty_cells = [(i, j) for i in range(9) for j in range(9) if puzzle[i][j] == 0]
```

3) Izgaranın her bir 3×3 alt karesi de 1'den 9'a kadar olan sayıları yalnızca bir kez içermelidir.

```
For i in range(9):

      if puzzle[row][i] == num or puzzle[i][col] == num: return False

   start_row, start_col = 3 * (row // 3), 3 * (col // 3)

   for i in range(3):

      for j in range(3):

         if puzzle[start_row + i][start_col + j] == num:

            return False
```

2. GENETİK ALGORİTMANIN KODLANMASI

Genetik algoritmaların temel işlemleri sırayla seçim, çaprazlama, ve mutasyon operatörleridir. Bu temel genetik operatörler, genetik algoritmaların bir problemi çözerken popülasyonu evrimleşerek daha iyi çözümler bulmasına yardımcı olmaktadır. Bu operatörlerin yanında uyumluluk fonksiyonu çalıştırılarak bulduğumuz çözümlerin gerçek çözüme yaklaşmasını ölçmekteyiz.

2.1 Uyumluluk Fonksiyonu

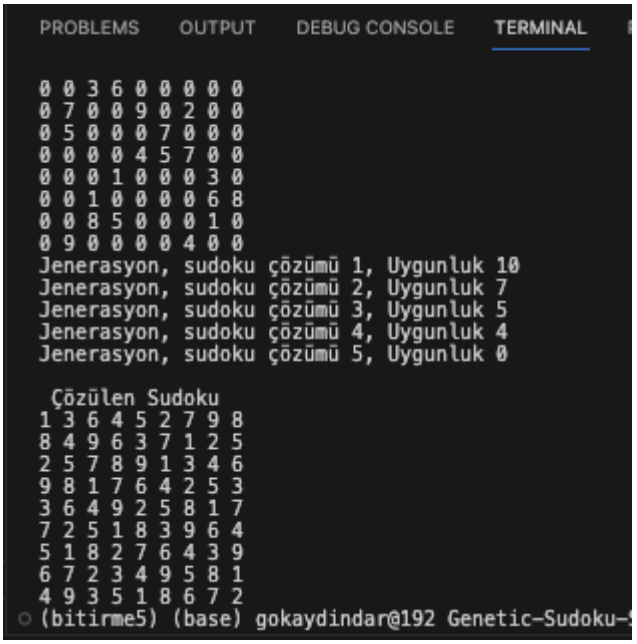
Uyumluluk fonksiyonu, her bireyin ne kadar iyi bir çözüm temsil ettiğini ölçen ve genetik algoritmanın doğru çözümleri bulmasına rehberlik eden bir ölçüt fonksiyondur. Sudoku problemimizde uyumluluk fonksiyonu, bir bireyin bir Sudoku çözümü olma derecesini ölçer. Amaç, uygun bir sudoku çözümüne sahip bireyleri belirleyerek bunları evrimleştirme sürecinde öne çıkarmaktadır. Fonksiyon, çeşitli kriterlere dayanarak bir Sudoku çözümünün ne kadar doğru olduğunu ölçmektedir. Satır ve sütunlarda herhangi bir tekrarlanan rakam uygunluk puanını artırır.

Fitness fonksiyonu ayrıca, her bir satırın, sütunun ve 3e3 bloğun toplamının 45 e eşit olup olmadığını kontrol eder. Bu, her bir rakamın yalnızca bir kez ve eksiksiz bir şekilde bulunduğundan emin olunmasını sağlar. Eğer toplam 45 e eşit değilse uygunluk puanı artar. Daha düşük uygunluk puanları, çözümün daha iyi olduğunu gösterirken, puan yükseldikçe sonuç kötüleşmektedir.

```
def fitness(puzzle, partial_puzzle):
    conflicts = 0
    for i in range(9):
        row = set(puzzle[i])
        column = set(puzzle[j][i] for j in range(9))
        block = set()
        for j in range(9):
            block_row, block_col = 3 * (i // 3), 3 * (j // 3)
            block.add(puzzle[block_row + (j // 3)][block_col + (j % 3)])


        # satırda 1-9 arası rakamların kontrolu
        if set(range(1, 10)) != row:
            conflicts += 1
         # sütünda 1-9 arası rakamların kontrolu
        if set(range(1, 10)) != column:
            conflicts += 1
        for i in range(9):
            if sum(puzzle[i]) != 45 or sum(puzzle[j][i] for j in range(9)) != 45:
                conflicts += 1


    return conflicts
```



Şekil 2.1 Fitness Değerinin Düşmesi

2.2. Çaprazlama Fonksiyonu

Çaprazlama fonksiyonu iki ebeveyn kromozomunun birleştirilerek yeni bir çocuk kromozomu oluşturulmasını sağlar. Bu işlem, Sudoku bulmacasını bir 3x3 alt kare matrisi olarak düşündüğümüzde, her alt kare satırının çocuk kromozomunda çakışmasız olması prensibine dayanmaktadır. İlk olarak, çocuk kromozomu, birinci ebeveynin kromozomunun bir kopyası olarak başlatılır. Bu durumda, çocuk kromozomu ilk aşamada birinci ebeveynin genetik materyalini içermektedir. Boş hücrelerin konumlarını belirlemek üzere, kısmi çözümdeki boş hücrelerin koordinatları bir liste içerisinde toplanır ve bu liste rastgele karıştırılır sonrasında her bir boş hücre için, rastgele bir seçim yapılır. Eğer bu seçim doğru ise, çocuk kromozomunun ilgili hücresi, birinci ebeveynin aynı konumdaki hücresinin değeri ile doldurulur. Ancak seçim yanlışsa çocuk kromozomunun ilgili hücresi ikinci ebeveynin aynı konumdaki hücresinin değeri ile doldurulur. İki ebeveynin genetik materyali birleştirilerek çocuk kromozomu oluşturulur ve genetik algoritmanın popülasyonu çeşitlenmiş olur.

```
def crossover(parent1, parent2, partial_puzzle):

   child = [row[:] for row in parent1]

   empty_cells = [(i, j) for i in range(9) for j in range(9) if
partial_puzzle[i][j] == 0]

   random.shuffle(empty_cells)

   for (i, j) in empty_cells:

       if random.choice([True, False]):

           child[i][j] = parent1[i][j]

       else:

           child[i][j] = parent2[i][j]

   return child
```

2.3 Mutasyon Fonksiyonu

Mutate fonksiyonu sudoku çözüm adaylarını mutasyona uğratan fonksiyondur. Fonksiyon belirli bir bireyin bir kopyasını oluşturur (mutasyonu uygulanmış birey), ardından bu kopyada rastgele seçilen iki hücrede değişiklik yapar. Her bir satır için, mutasyona uğrayacak hücreler belirlenir ve bu hücreler rastgele sırayla karıştırılır. Her bir hücre için, o hücreye yerleştirilebilecek geçerli sayılar belirlenir. Bu sayılar, o hücrede zaten bulunmayan ve Sudoku kurallarına uygun olan sayılardır. Belirlenen geçerli sayılardan rastgele bir tanesi seçilerek, mutasyona uğramış bireyin ilgili hücresine yerleştirilir. Bu işlem, her satır için en fazla iki hücrede tekrarlanır. Makalede İki farklı mutasyon operasyonu kullanılmaktadır. İlk mutasyon, kromozomun bir genine mutationProb1 olasılığıyla uygulanır. Eğer bu mutasyon bir gene uygulanıyorsa, aynı satırdaki başka bir rastgele önceden tanımlanmamış gen seçilir ve bu iki genin değerleri değiştirilir. Daha sonra, yeni kromozomun bizim kısıtlamalarımıza göre geçerli olup olmadığını kontrol ederiz. Eğer yeni kromozom geçerli değilse, aynı değerlere sahip iki geni aynı alt kare satırının diğer iki matris satırında değiştirerek geçerli bir kromozom elde ederiz. İkinci mutasyon, her bir alt kare satırına mutationProb2 olasılığıyla uygulanır. Bir mutasyon bir alt kare satırına gerçekleştiğinde, o satır tamamen yeni bir şekilde oluşturulan bir satır ile değiştirilir. Ancak kodda tamamen olasılık fonksiyonları yerine rastgelelik kullandım.

Şekil 2.2 Mutasyona Uğrayan Aday Çözümler

```
def mutate(individual, partial_puzzle):

    mutated_individual = [row[:] for row in individual]

   for i in range(9):

       empty_cells = [(i, j) for j in range(9) if partial_puzzle[i][j] == 0]

       random.shuffle(empty_cells)

       for (i, j) in empty_cells[:2]:  # Mutate at most 2 cells in each row

           valid_numbers = [num for num in range(1, 10) if
is_valid_move(mutated_individual, i, j, num)]

           if valid_numbers:

               mutated_individual[i][j] = random.choice(valid_numbers)

   #print(mutated_individual)

   return mutated_individual
```

2.4 Kural Kontrolü

Aday çözümün partial ızgaraya uygulanırken belirli bir sayının yerleştirilip yerleştirilemeyeceğini belirlemek amacıyla kural kontrolü yapıyoruz bunu is_valid_move fonksiyonu gerçekleştiriyor. İlk olarak, fonksiyon belirli bir satırdaki diğer hücrelerde aynı sayının bulunup bulunmadığını kontrol eder eğer aynı sayı başka bir hücrede bulunuyorsa belirtilen hücreye bu sayıyı yerleştirmek geçersiz işlem olarak kabul edilir benzer şekilde, fonksiyon aynı sayının bulunup bulunmadığını kontrol etmek üzere belirli bir sütunu tarar eğer aynı sayı başka bir hücrede bulunuyorsa bu sayıyı belirtilen hücreye yerleştirmek geçersiz işlemdir. Ayrıca, Sudoku tahtasındaki hücrelerin 3x3'lük bloklara ayrılmasını dikkate alır. Aynı sayının aynı alt kare bloğu içinde başka bir hücrede bulunup bulunmadığını kontrol eder. Eğer aynı sayı başka bir hücrede bulunuyorsa bu sayıyı belirtilen hücreye yerleştirmek geçersiz kabul edilir. Bu kontrollerin hiçbiri ihlal edilmiyorsa, hamle geçerli kabul edilir ve True değeri döner. Aksi halde, hamle geçersiz kabul edilir ve False değeri döner.

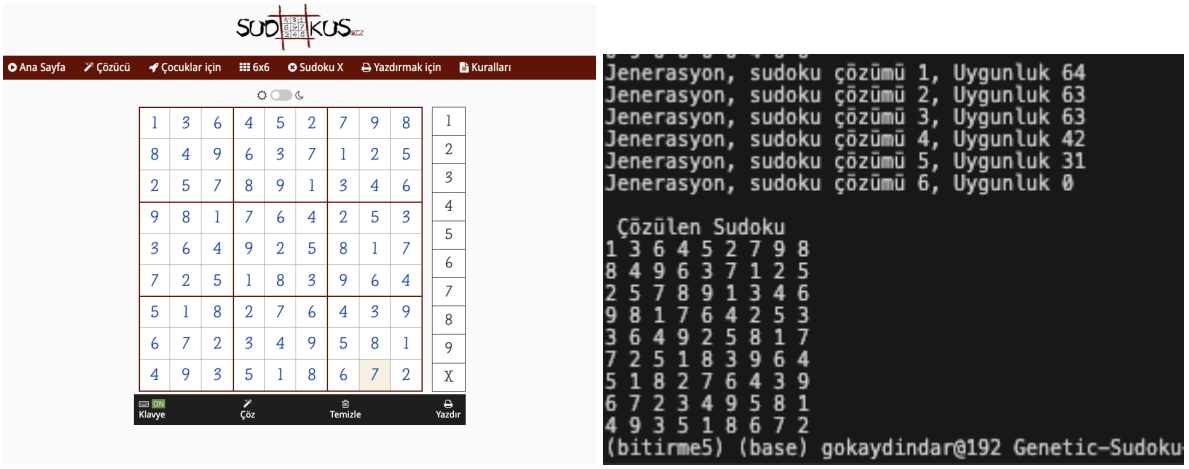
3. GENETİK ALGORİTMANIN UYGULANMASI

Öncelikle genetik algoritmamızı çalıştırmak için iki fonksiyona daha ihtiyacımız bulunmakta bunlardan generate_individual fonksiyonu sudoku tahtasındaki boş hücrelere rastgele sayılar yerleştirerek farklı başlangıç aday çözümleri oluşturur rastgele biçimde. Başlangıç populasyonu oluşturulduktan sonra genetik algoritma bu bireyleri çaprazlama, mutasyon ve uygunluk değerlerine göre sıralama gibi operasyonlarla evrimleştirir. Yeni nesil bireyler, önceki neslin en iyi bireylerinden türetilir ve rastgelelik ile çeşitlilikte korunmaya çalışınır. başlangıçta generate_individual ile bir grup sayıdan oluşan bireyler aday çözümler oluşturduk. İlk oluşturulan bu bireylerin ne kadar iyi olduğunu ölçmek için bir "uygunluk" değeri kullanıyoruz. Sonra yeni jenerasyonlar oluşturuyoruz. Aday çözümlerde en iyi uygunluk değeri 0 ise süreci tamamlarız. Eğer mükemmel bir çözüm bulunamamışsa, yeni bir jenerasyon yaratırız. Bunun için mevcut bireyler arasından rastgele seçilmiş iki ebeveyni alırız. Bu ebeveynlerin genetik özelliklerini birleştirerek ve rastgele değişiklikler yaparak yeni bireyler üretiriz. Bu yeni bireyler, bir sonraki jenerasyona eklenir. Bu işlem, belirli bir jenerasyon sayısına veya mükemmel bir çözüm bulana kadar devam eder. Süreç tamamlandığında, en iyi uygunluğa sahip birey ve bu bireyin Sudoku çözümü elde edilir.

```
def genetic_algorithm(partial_puzzle, population_size, generations):
            population = [generate_individual(partial_puzzle) for _ in
range(population_size)]
   for generation in range(generations):
       population = sorted(population, key=lambda x: fitness(x, partial_puzzle))
       best_fitness = fitness(population[0], partial_puzzle)
               print(f"Jenerasyon, sudoku çözümü {generation + 1}, Uygunluk
{best_fitness}")
       if best_fitness == 0:
           break
       new_population = []
       for _ in range(population_size // 2):
           parent1 = random.choice(population[: population_size // 2])
           parent2 = random.choice(population[: population_size // 2])
           child1 = crossover(parent1, parent2, partial_puzzle)
           child2 = crossover(parent1, parent2, partial_puzzle)
              new_population.extend([mutate(child1, partial_puzzle), mutate(child2,
partial_puzzle)])
       population = new_population
   best_solution = sorted(population, key=lambda x: fitness(x, partial_puzzle))[0]
   return best_solution
```

3.1. Genetik Algoritmanın Test Edilmesi

Genetik algoritmamızı test etmek için internet üzerinde bulunan sudoku çözme servisleri kullanılmıştır. 4 adet farklı zorlukta sudoku problemleri öncelikli olarak internet sitesinde sonuçları incelenmiş daha sonra partial biçimde genetik algoritma kodumuza yüklenmiştir. Kolay sudokular ilk 3 jenerasyon içerisinde 0 uygunluk(çözüm) e ulaşmaktadır. Daha zor problemler ise 10 - 12 jenerasyon içerisinde çözüme ulaşmaktadır. Popülasyon büyüklüğü 50, jenerasyon boyutu 1000 olarak girilmiştir.



Şekil 3.1 *İnternet Sitesinde Doğrulanan Çözüm ve Sağ Tarafta Uygulamamızın Bulduğu Çözüm*

4. SONUÇ

Makalede tartışılan genetik algoritma yöntemi ile sudoku bulmacalarının çözümü başarıyla gerçekleştirilmiştir internet üzerinden çözümü bilinen sudokular ile test edilmiştir. Makalede kodlama detaylarına odaklanılmamış olmasına rağmen genetik algoritma yaklaşımı yazdığımız kodda başarıyla uygulanıp githubta paylaşılmıştır. Genetik algoritmaların pratik avantajları canlı gözlemlenmiş olup bu yaklaşım ile sudoku çözülmesi konusunda bilgi sahibi olunmuştur.

5. KAYNAKLAR

[1] T. Mantere and J. Koljonen, “Solving and rating sudoku puzzles with genetic algorithms,” in New Developments in Artificial Intelligence and the Semantic Web, Proceedings of the 12th Finnish Artificial Intelligence Conference STeP, pp. 86–92, 2006.

[2] Road To Machine Learning, Solving Sudoku puzzles with Genetic Algorithm, https://nidragedd.github.io/sudoku-genetics/

[3] Genetic Algorithms and Sudoku, Dr. John M. Weiss Department of Mathematics and Computer Science South Dakota School of Mines and Technology (SDSM&T) Rapid City, SD 57701-3995 john.weiss@sdsmt.edu MICS 2009

[4] Solving Sudoku with machine learning methods - genetic algorithm and simulated annealing,https://jiayiwu.me/blog/2022/03/13/solving-sudoku-with-machine-learning-methods-genetic-algorithm-and-simulated-annealing.html