Prof. Dr. Mine Elif KARSLIGİL BLM3021 Algoritma Analizi

**Son Teslim tarihi:** **10.01.2025**

Ödev #4

Oğuzhan Topaloğlu

Ç19052025 – Grup 1

*Bilgisayar Mühendisliği Bölümü,*

*Elektrik-Elektronik Fakültesi,*

*Yıldız Teknik Üniversitesi*

****

Istanbul, 2025

**Video Linki:** [**https://youtu.be/DKRlGfG6\_t8**](https://youtu.be/DKRlGfG6_t8)

|  |
| --- |
| **İÇERİK** |
| 1. Problemin Çözümü |
| 2. Karşılaşılan Sorunlar |
| 3. Karmaşıklık Analizi |
| 4. Ekran Görüntüleri |
|  |
|  |
|  |
|  |
|  |

**1. Problemin Çözümü**

Ödevde N-Queens problemini çözmek için dört farklı algoritma implementasyonu kodlanmıştır: Brute Force, Optimized\_1, Optimized\_2, ve Backtracking. Amaç, her algoritmanın çözüm üretme süresi ve verimliliğini karşılaştırmaktır. N-Queens problemi, N x N boyutundaki bir tahtada N adet vezirin yerleştirilmesi gereken bir problemdir. Her vezir bir satıra ve sütuna yerleştirilmelidir, aynı zamanda hiçbir vezir diğerini tehdit etmemelidir.

Algoritmaların Açıklamaları:

1- Brute Force

Tüm olası yerleştirme kombinasyonları (permütasyonlar) hesaplanır ve her bir kombinasyonun geçerliliği kontrol edilir. Bu basit ve doğrudan bir yaklaşımdır ancak çözüm bulma süresi çok uzun olabilir. Özellikle N büyüdükçe hesaplar dakikaları ve saatleri aşmaktadır. Bunun nedeni permütasyon sayısının faktöriyel bir şekilde büyümesidir.

2- Optimized\_1

Bu yöntemde satır satır ilerlenerek her bir satıra bir vezir yerleştirilir. Her vezir yerleştirildikten sonra, önceki satırlarda yerleştirilen vezirlerle çatışma olup olmadığı kontrol edilir. Bu yöntemin geçerlilik kontrolü, yalnızca önceki satırlardaki yerleşimlere dayalıdır, böylece hesaplama maliyeti düşer.

3- Optimized\_2

Bu yöntem Optimized\_1'e benzer ancak bir ek optimizasyon içerir. Bu ek optimizasyon da satırların dışında hangi satırların dolu olduğunun bir dizide tutulmasıdır. Bu algoritmada önceden işaretlenmiş sütun bilgileri sayesinde gereksiz hesaplamalar engellenir. Optimized\_1'e göre daha fazla bellek kullanılır ancak hızda gelişmeler elde edilir.

4- Backtracking

Vezir yerleştirilirken her adımda, geçerli yerleştirmenin geçerliliği kontrol edilir. Geçerli değilse, algoritma önceki adımlara geri dönerek yeni bir yerleşim dener. Bu çözümün avantajı çözüm bulunduğunda algoritma hemen sonlanmasıdır. Bunun sayesinde bir sürü gereksiz adımlar atlanır. Diğer yöntemlere göre daha karmaşıktır ancak çok daha verimli çalışır.

**2. Karşılaşılan Sorunlar**

Ödev sırasında en çok optimized\_1 ve optimized\_2 implementasyonunda sorun yaşadım. Bunun nedeni backtracking ve brute force için internette bir sürü kaynak olmasına rağmen optimized\_1 ve optimized\_2 için olmamasıydı.

Optimized\_2’yi tamamladıktan sonra for döngülerinin sırasını karıştırdığım için sütunları kullanarak ek inceleme yapmamayı efektif bir şekilde kullanamıyordum. Bu yüzden de optimized\_2’de optimized\_1’den daha yavaş bir sonuç elde ediyordum. Bu hatamı düzeltmem bayağı zaman aldı.

Ayrıca print etme sürecinde de sıkıntılar yaşadım. Bir sürü function signature’ı değiştiğinden refactoring zaman aldı. Sonunda helper fonksiyonlar tanımlayarak refactoring’i bitirmeyi başardım.

**3. Karmaşıklık Analizi**

1- Brute Force Algoritması

Brute Force algoritması, problemi çözmek için tüm olası yerleşim permütasyonlarını hesaplar. Bu da tahta üzerindeki N \* N hücrede her veziri yerleştirmeyi dener. Permütasyonların sayısı N! olduğundan, her hücre için N farklı yerleşim denemek gerekmektedir. Bu işlemin her biri N! kadar tekrarlandığı için zaman karmaşıklığı O(N!)'dir.

2- Optimized\_1 Algoritması

Optimized\_1 algoritması, her bir satırda yerleştirilecek veziri, daha önce yerleştirilen vezirlerin sütunlarına ve çaprazlarına bakarak geçerli yerleşimlerden seçer. Bu, yerleşimlerin sayısını azaltır ve Brute Force'tan daha verimli bir çözüm sunar. Ancak yine de her satırda geçerli yerleşimleri kontrol etmek için N \* N kadar işlem yapılması gerektiği için zaman karmaşıklığı O(N^2)'dir.

3- Optimized\_2 Algoritması

Optimized\_2, Optimized\_1 ile benzer şekilde çalışır ancak sütun kontrolü de işin içine eklenir. Bu ek kontrol sayesinde, average case’de Optimized\_1'den daha hızlı çalışır. Ancak genel karmaşıklık hala O(N^2)'dir, çünkü her satırda yine N farklı kontrol yapılır ve bu işlem her satır için tekrarlanır.

4- Backtracking Algoritması

Backtracking algoritması, her bir satırda yerleştirilen vezirlerin sütunlarını ve çaprazlarını işaretler ve geçersiz yerleşimleri hızlıca dışlar. Bu sayede sadece geçerli çözümler üzerinde işlem yapılır. Her satırda geçerli yerleşimlerin kontrol edilmesi için O(N) işlem yapılır. Ancak her satır için bu işlem tekrarlanır, bu da teorik olarak O(N^2) karmaşıklığa yol açar. Bununla birlikte, geçersiz yerleşimler anında dışlandığı için büyük N'lerde pratikte daha hızlı çalışabilir ve karmaşıklık daha düşük seviyelere inebilir.

**4. Ekran Görüntüleri**





















