0/1 Knapsack Problemi İçin Optimal ve Yaklaşık Çözüm Algoritmalarının Analizi

Muhammed Umut Şıbara
Öğrenci No: 222804007
Celal Bayar Üniversitesi
Algoritma Analizi ve Tasarımı Dersi

Haziran 2025

Özet

Bu çalışmada, kombinatoryal optimizasyonun en temel problemlerinden biri olan 0/1 Knapsack (Sırt Çantası) Problemi için farklı algoritmik yaklaşımlar geliştirilmiş, uygulanmış ve karşılaştırılmıştır. Çalışmanın temel amacı, belirli bir kapasite kısıtı altında toplam değeri maksimize etmeyi hedefleyen bu NP-Zor (NP-Hard) problem için kesin (optimal) ve yaklaşık (sezgisel) çözüm yöntemlerinin performansını, verimliliğini ve pratik uygulanabilirliklerini analiz etmektir.

Çözüm sürecinde iki ana kategoriye odaklanılmıştır. İlk olarak, optimal sonucu garanti eden **Dinamik Programlama (DP)** ve **Branch and Bound (B&B)** gibi kesin çözüm algoritmaları incelenmiştir. İkinci olarak, optimalite garantisi olmaksızın çok daha hızlı bir şekilde yüksek kaliteli çözümler üretmeyi amaçlayan **Basit Açgözlü**, **Geliştirilmiş Açgözlü**, **Yerel Arama (Local Search)** ve **Genetik Algoritma** gibi sezgisel ve meta-sezgisel yöntemler uygulanmıştır.

Algoritmalar, 40, 300, 1000 ve 10000 eleman içeren dört farklı boyuttaki standart test verisi üzerinde Python programlama dili kullanılarak test edilmiştir. Deneysel sonuçlar, Dinamik Programlama'nın büyük kapasiteli problemlerde yaşadığı "Out of Memory" sorununu ve Branch and Bound algoritmasının zorlu veri setlerinde sergilediği pratik limitleri ortaya koymuştur. Sezgisel algoritmaların ise optimal çözüme ne kadar yaklaşabildikleri, hız ve sonuç kalitesi arasındaki takas (trade-off) detaylı bir şekilde analiz edilmiştir. Bu çalışma, teorik algoritmaların pratik problemlere uygulandığında karşılaşılan zorlukları ve farklı çözüm stratejilerinin ne zaman tercih edilmesi gerektiğini gösteren kapsamlı bir vaka analizi sunmaktadır.

İçindekiler

Şekil Listesi

Tablo Listesi

Bölüm 1

Giriş

Optimizasyon, mühendislikten finansa, lojistikten bilgisayar bilimine kadar sayısız alanda en verimli ve etkili kararları alabilmek için kullanılan temel bir disiplindir. Bu disiplinin kalbinde yer alan en klasik ve öğretici problemlerden biri de 0/1 Knapsack (Sırt Çantası) Problemi'dir. Problem, basit bir senaryo üzerinden karmaşık bir karar verme sürecini modeller: Sınırlı taşıma kapasitesine sahip bir sırt çantası olan bir gezgin, her birinin kendi ağırlığı ve değeri olan bir dizi eşya arasından hangilerini çantasına koyacağına karar vermelidir. Amaç, çantanın kapasitesini aşmadan, taşınan eşyaların toplam değerini maksimum seviyeye çıkarmaktır. "0/1" kısıtı, her bir eşya için kararın ikili olduğunu belirtir; eşya ya tamamen alınır (1) ya da tamamen bırakılır (0).

Matematiksel sadeliğine rağmen 0/1 Knapsack Problemi, çözülmesi zor problemlerin sınıflandırıldığı **NP-Zor (NP-Hard)** kategorisinde yer alır. Bu, problemin boyutları (özellikle eşya sayısı) arttıkça, olası tüm kombinasyonları deneyerek (Brute Force) optimal çözümü bulmanın pratik olarak imkansız hale geldiği anlamına gelir. n adet eşya için 2^n adet olası çözüm kombinasyonu bulunur ve bu sayı, mütevazı n değerleri için bile astronomik seviyelere ulaşır. Bu zorluk, problemi çözmek için daha akıllı ve verimli algoritmaların geliştirilmesini zorunlu kılmıştır.

Bu çalışma, 0/1 Knapsack Problemi'ne yönelik iki temel felsefeyi ele almaktadır:

1. **Kesin (Optimal) Çözüm Yaklaşımları:** Optimal sonucu bulmayı garanti eden ancak yüksek hesaplama maliyetine sahip olabilen Dinamik Programlama ve Branch and Bound gibi algoritmalar.

2. **Yaklaşık (Sezgisel) Çözüm Yaklaşımları:** Optimalite garantisinden feragat ederek çok daha hızlı bir şekilde "yeterince iyi" çözümler üreten Açgözlü (Greedy) algoritmalar ve Genetik Algoritma gibi meta-sezgisel yöntemler.

Bu raporun amacı, belirtilen algoritmaları farklı ölçeklerdeki veri setleri üzerinde uygulayarak performanslarını karşılaştırmak, her bir yöntemin güçlü ve zayıf yönlerini

analiz etmek ve teorik karmaşıklıklarının pratik sonuçlara nasıl yansıdığını göstermektir. Raporun ilerleyen bölümlerinde, öncelikle problemle ilgili yapılmış önemli akademik çalışmaların bir özeti sunulacak (Bölüm 2), ardından kullanılan tüm algoritmaların teorik altyapısı detaylandırılacak (Bölüm 3), elde edilen deneysel sonuçlar tablolar ve grafiklerle analiz edilecek (Bölüm 4) ve son olarak genel bir tartışma ile çalışma neticelendirilecektir (Bölüm 5).

Bölüm 2

Literatür Özeti

0/1 Knapsack Problemi, 19. yüzyıldan beri üzerinde çalışılan ve kombinatoryal optimizasyon alanında bir köşe taşı olarak kabul edilen bir problemdir. Zengin tarihi boyunca, problemin çözümü için sayısız makale, kitap ve tez yayınlanmıştır. Bu bölümde, problemi ve çözüm yöntemlerini şekillendiren bazı temel ve etkili akademik çalışmalar özetlenmektedir.

- **Dantzig, G. B. (1957). "Discrete-Variable Extremum Problems."** Lineer programlamanın babası olarak kabul edilen George Dantzig, bu öncü çalışmasında Knapsack Problemi'nin sürekli (continuous) versiyonunu ele almıştır. Bu versiyonda, eşyaların parçalara ayrılmasına izin verilir. Dantzig, sürekli Knapsack Problemi'nin, eşyaların değer/ağırlık oranlarına göre sıralanıp en kârlı olandan başlanarak çantanın doldurulmasıyla optimal olarak çözülebileceğini göstermiştir. Bu basit "açgözlü" yaklaşım, daha karmaşık olan 0/1 Knapsack problemi için geliştirilen birçok sezgisel ve B&B gibi kesin çözüm algoritmalarında temel bir alt rutin olarak kullanılmaktadır.

- **Bellman, R. (1957). *Dynamic Programming.*** Richard Bellman, bu eseriyle Dinamik Programlama paradigmasını dünyaya tanıtmıştır. Bellman'ın "Optimalite Prensibi" (Principle of Optimality), büyük bir problemin optimal çözümünün, o problemi oluşturan alt problemlerin optimal çözümlerini içerdiği fikrine dayanır. 0/1 Knapsack Problemi, bu prensibin en başarılı uygulama alanlarından biridir. Problemin özyineli (recursive) yapısı, DP tablosu kullanılarak verimli bir şekilde çözülebilmekte ve bu yöntem, problemin sözde-polinomsal zamanda ($O(nW)$) optimal çözümünü veren standart bir teknik haline gelmiştir.

- **Horowitz, E., & Sahni, S. (1974). "Computing partitions with applications to the knapsack problem."** Bu etkili makale, "meet-in-the-middle"

olarak da bilinen, kaba kuvvet aramasını önemli ölçüde hızlandıran bir teknik sunmuştur. Yazarlar, eşya setini ikiye bölerek her bir yarı için tüm olası alt çözümleri üretmiş ve ardından bu iki çözüm setini birleştirerek optimal sonucu bulmuşlardır. Bu yaklaşım, $O(2^n)$ olan kaba kuvvet karmaşıklığını $O(2^{n/2})$ seviyesine indirerek, orta boyutlardaki ($n \approx 40-50$) problemlerin optimal çözümünü pratik hale getirmiştir. Bu çalışma, üssel zamanda çalışan algoritmaların pratik sınırlarını genişletme konusunda önemli bir adım olarak kabul edilir.

- **Garey, M. R., & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness.*** Bilgisayar biliminin en temel eserlerinden biri olan bu kitap, NP-Zorluk teorisini standartlaştırmıştır. Yazarlar, 0/1 Knapsack Problemi'ni resmi olarak NP-Zor problemler listesine dahil etmiş ve bu problemin (ve benzerlerinin) neden polinomsal zamanda çalışan bir çözüm algoritmasının bulunmasının olası olmadığını matematiksel olarak açıklamışlardır. Bu kitap, Knapsack problemi için neden sezgisel ve yaklaşık çözüm algoritmalarının bu kadar önemli olduğunu anlamak için temel bir kaynaktır.

- **Martello, S., & Toth, P. (1990). *Knapsack Problems: Algorithms and Computer Implementations.*** Bu kitap, Knapsack problemleri üzerine yazılmış en kapsamlı ve en çok atıf alan eserlerden biridir. Martello ve Toth, problemin tüm varyantlarını (0/1, Bounded, Unbounded, Multi-dimensional vb.) detaylıca inceler. Kitap, Dinamik Programlama ve Branch and Bound için son derece verimli ve optimize edilmiş algoritmalar sunar. Özellikle B&B için geliştirdikleri üst sınır (upper bound) hesaplama teknikleri ve indirgeme (reduction) stratejileri, günümüzde bile birçok modern Knapsack çözücüsünün temelini oluşturmaktadır.

- **Lawler, E. L. (1979). "Fast approximation algorithms for knapsack problems."** Optimal çözüme ulaşmanın zor olduğu durumlarda, yaklaşık çözümlerin ne kadar değerli olduğunu gösteren önemli bir çalışmadır. Lawler, bu makalesinde, kullanıcının belirlediği bir hata payı (ϵ) dahilinde optimale yakın sonuçları garanti eden ve çalışma süresi hem problemin boyutuna hem de $1/\epsilon$'a polinomsal olarak bağlı olan FPTAS (Fully Polynomial-Time Approximation Scheme) kavramını Knapsack problemi için popülerleştirmiştir. Bu, hız ve kalite arasındaki takasın matematiksel bir çerçeveye oturtulmasını sağlamıştır.

- **Khuri, S., Bäck, T., & Heitkötter, J. (1994). "The zero/one multiple knapsack problem and genetic algorithms."** Bu çalışma, Knapsack probleminin çözümünde Genetik Algoritmaların (GA) etkinliğini gösteren ilk ve etkili

uygulamalardan biridir. Yazarlar, bir çözümü nasıl bir "kromozom" olarak temsil edeceklerini, uygunluk fonksiyonunu nasıl tasarlayacaklarını ve çaprazlama/mutasyon operatörlerini probleme nasıl uyarlayacaklarını detaylıca açıklamışlardır. Bu makale, GA'nın karmaşık kombinatoryal optimizasyon problemlerinde nasıl güçlü bir arama ve keşif aracı olarak kullanılabileceğini göstermesi açısından önemlidir.

- **Pisinger, D. (2005). "Where are the hard knapsack problems?".** Bu ilginç makale, hangi tür Knapsack veri setlerinin algoritmalar için "zor" olduğunu araştırır. Pisinger, özellikle değer ve ağırlıkların birbiriyle güçlü bir şekilde korele olduğu durumlarda, Branch and Bound gibi algoritmaların performansının önemli ölçüde düştüğünü göstermektedir. Bu çalışma, bizim `ks_10000_0` veri setinde yaşadığımız aşırı yavaşlama probleminin teorik arka planını açıklamakta ve bu tür zorlu örneklerin algoritma testlerinde neden önemli olduğunu vurgulamaktadır.

Bu temel çalışmalar, raporumuzda ele aldığımız algoritmik yaklaşımların seçiminin tesadüfi olmadığını, aksine problemin teorik ve pratik zorluklarına yönelik on yıllardır süren akademik bir birikime dayandığını göstermektedir.

Bölüm 3

Kullanılan Yöntemler ve Algoritmalar

Bu çalışmada, 0/1 Knapsack Problemi'nin çözümü için hem optimal sonucu garanti eden kesin (exact) algoritmalar hem de pratik ve hızlı sonuçlar üreten yaklaşık (heuristic) çözüm yöntemleri incelenmiş ve uygulanmıştır. Bu bölüm, kullanılan her bir algoritmanın teorik temelini, çalışma prensibini, algoritmik karmaşıklığını ve Python dilinde gerçekleştirilen uygulamasını detaylı bir şekilde açıklamaktadır. Yaklaşımlar, "Optimal Çözüm Algoritmaları" ve "Yaklaşık Çözüm Algoritmaları" olmak üzere iki ana başlık altında sunulmuştur.

3.1 Optimal Çözüm Algoritmaları

Bu kategorideki algoritmaların temel hedefi, tüm çözüm uzayını akıllıca tarayarak matematiksel olarak kanıtlanabilir en iyi (optimal) sonucu bulmaktır. Bu yöntemler, çözüm kalitesinden ödün vermezler ancak problemin boyutlarına bağlı olarak yüksek hesaplama maliyetine sahip olabilirler.

3.1.1 Dinamik Programlama (DP)

Dinamik Programlama, Richard Bellman'ın Optimalite Prensibi'ne dayanan ve büyük bir problemi, kendini tekrar eden daha küçük alt problemlerin çözümlerini kullanarak çözen güçlü bir tekniktir. 0/1 Knapsack problemi, DP'nin bu yapısına mükemmel bir şekilde uymaktadır.

Teorik Altyapı ve Formülasyon

DP yaklaşımında, n adet eşya ve W kapasiteli bir çanta için $DP[i][w]$ şeklinde iki boyutlu bir tablo oluşturulur. Bu tablodaki her bir hücre, şu alt problemi cevaplar:

"İlk i adet eşyayı kullanarak, w kapasiteli bir çanta ile elde edilebilecek maksimum değer nedir?"

Bu tabloyu doldurmak için kullanılan özyineli (recursive) formül şu şekildedir: i. eşyanın değeri v_i ve ağırlığı w_i olmak üzere;

$$DP[i, w] = \begin{cases} DP[i-1, w] & \text{eğer } w_i > w \text{ (eşya çantaya sığmıyorsa)} \\ \max(DP[i-1, w], v_i + DP[i-1, w-w_i]) & \text{eğer } w_i \leq w \text{ (eşya çantaya sığıyorsa)} \end{cases}$$

Bu formül, her bir eşya için iki temel kararı değerlendirir:

- **Eşyayı almamak:** Bu durumda, maksimum değer, bir önceki alt problemde ($i-1$ eşya ile) aynı kapasiteyle (w) elde edilen değerdir: $DP[i-1, w]$.
- **Eşyayı almak:** Bu durumda, maksimum değer, mevcut eşyanın değeri (v_i) ile, bu eşyayı aldıktan sonra çantada kalan kapasiteye ($w-w_i$) önceki eşyalarla ($i-1$) sığdırılabilen maksimum değerin toplamıdır: $v_i + DP[i-1, w-w_i]$.

Algoritma, bu iki seçenekten daha yüksek değeri vereni seçerek tabloyu sistematik bir şekilde doldurur. Tüm tablo dolduğunda, $DP[n][W]$ hücresi problemin nihai optimal çözümünü içerir. Çözüme hangi eşyaların dahil edildiğini bulmak için ise bu tablonun sonundan başına doğru bir **geri izleme (backtracking)** işlemi yapılır.

Karmaşıklık Analizi

- **Zaman Karmaşıklığı:** Algoritma, $n \times W$ boyutlarındaki tablonun her bir hücresini sabit zamanda doldurur. Bu nedenle zaman karmaşıklığı $O(nW)$'dir.
- **Alan (Bellek) Karmaşıklığı:** Tablonun kendisi $O(nW)$'lik bir bellek alanı gerektirir.

Bu karmaşıklık, "sözde-polinomsal" (pseudo-polynomial) olarak adlandırılır. Çünkü çalışma süresi, girdinin sayısal değerine (W) polinomsal olarak bağlıdır. Bu durum, W değeri çok büyüdüğünde algoritmanın pratik olarak kullanılamaz hale gelmesine neden olur.

Python Uygulaması

Aşağıda, Dinamik Programlama yaklaşımını ve ilerleme bildirimlerini içeren tam Python kodu verilmiştir.

```python
# -*- coding: utf-8 -*-
# Dosya Ad : dp_solver.py
# A klama: 0/1 Knapsack problemini Dinamik Programlama ile zer .
#  alrken terminale detayl ilerleme durumu bildirir
    .

import sys
import time

def solve_with_dp(file_path):
    print("=" * 60)
    print(f"  leniyor : '{file_path}' (Algoritma: Dinamik Programlama
    )")
    print("=" * 60)

    try:
        print("[1/4]     Dosya okunuyor...")
        with open(file_path, 'r') as f:
            lines = f.readlines()
            num_items, capacity = map(int, lines[0].strip().split())
            values = [int(line.strip().split()[0]) for line in lines
    [1:] if line.strip()]
            weights = [int(line.strip().split()[1]) for line in lines
    [1:] if line.strip()]
    except Exception as e:
        print(f"HATA: Dosya i lenirken bir sorun olu tu! Detay: {e}"
    )
        return

    print("\n[2/4]     DP tablosu dolduruluyor...")
    start_time = time.time()
    dp = [[0 for _ in range(capacity + 1)] for _ in range(num_items +
    1)]

    for i in range(1, num_items + 1):
        val, wt = values[i-1], weights[i-1]
        for w in range(capacity + 1):
            if wt <= w:
                dp[i][w] = max(dp[i-1][w], val + dp[i-1][w - wt])
            else:
                dp[i][w] = dp[i-1][w]
        if i % (num_items // 10 or 1) == 0:
            print(f" ->  lerleme : %{(i / num_items) * 100:.0f}
    tamamland ...")
```

```
38
39      print("\n[3/4]        Geri izleme (backtracking) yap l yor...")
40      optimal_value = dp[num_items][capacity]
41      selected_items = []
42      w = capacity
43      for i in range(num_items, 0, -1):
44          if dp[i][w] != dp[i-1][w]:
45              selected_items.append(i)
46              w -= weights[i-1]
47      selected_items.reverse()
48      end_time = time.time()
49
50      # --- YEN   FORMAT    N      IKTI   OLU TURMA ---
51      binary_solution = ['0'] * num_items
52      for item_index in selected_items:
53          binary_solution[item_index - 1] = '1'
54
55      #   stenen   formatlar: "0,0,1,1" ve "3,4"
56      binary_output_str = ','.join(binary_solution)
57      item_list_output_str = ','.join(map(str, selected_items))
58
59      print("\n[4/4]         zm    Raporu:")
60      print("-" * 30)
61      print(f"    alma     S resi: {end_time - start_time:.4f} saniye")
62      print("\n--- Excel    in       ktlar     ---")
63      print(f"Optimal Value De eri: {optimal_value}")
64      print(f"Optimal     zm    : {binary_output_str}")
65      print(f"Optimal     zme     dahil edilen itemler: {
    item_list_output_str}")
66      print("=" * 60 + "\n")
67
68  if __name__ == "__main__":
69      if len(sys.argv) > 1:
70          for file_name in sys.argv[1:]:
71              solve_with_dp(file_name)
72      else:
73          print("Kullan m: python dp_solver.py <dosya_ad >")
```

Listing 3.1: Dinamik Programlama Çözücüsü - dp_solver.py

3.1.2 Branch and Bound (B&B)

Branch and Bound, DP'nin pratik olmadığı (özellikle bellek kısıtları nedeniyle) büyük ölçekli NP-Zor problemlerde optimal sonucu bulmak için kullanılan bir diğer kesin çözüm yöntemidir. Temel mantığı, tüm çözüm uzayını temsil eden bir durum ağacını (state-space tree) akıllıca keşfetmektir.

Teorik Altyapı ve Formülasyon

B&B, "böl ve yönet" stratejisini, umut vadetmeyen çözüm adaylarını sistematik olarak eleyen bir "budama" (pruning) mekanizması ile birleştirir.

1. **Durum Ağacı (State-Space Tree):** Problem, kök düğümden başlayan bir ağaç olarak modellenir. Ağacın her bir seviyesi, bir sonraki eşya için verilecek kararı temsil eder: eşyayı "al" (sol dal) veya "alma" (sağ dal). Kökden bir yaprak düğüme giden her yol, olası bir çözüme karşılık gelir.

2. **Sınır Hesaplama (Bounding):** Algoritmanın en kritik parçasıdır. Ağaçtaki herhangi bir düğüm (yani kısmi bir çözüm) için, o yoldan devam edildiğinde elde edilebilecek **potansiyel en iyi değerin bir üst sınırı (upper bound)** hesaplanır. Bu üst sınır, genellikle problemin daha basit bir versiyonunu çözerek elde edilir. Knapsack problemi için en yaygın ve etkili sınır hesaplama yöntemi, problemin sürekli (continuous) versiyonunu çözmektir:

   - Mevcut düğümdeki toplam ağırlık ve değer hesaplanır.
   - Kalan eşyalar, değer/ağırlık oranına göre sıralanır.
   - Çantada kalan kapasite, en kârlı eşyalardan başlanarak doldurulur. Çantaya tam sığmayan son eşyanın ise kesirli bir parçası alınıyormuş gibi varsayılarak potansiyel maksimum değer hesaplanır. Bu bize, gerçekte ulaşılamayacak ama iyi bir tahmin olan "optimistik" bir üst sınır verir.

3. **Budama (Pruning):** Algoritma, o ana kadar bulduğu en iyi geçerli çözümün değerini ($\text{`max}_p rofit\text{`}veya\text{`}lower_b ound\text{`})haf ßzasßndatutar.Yenibirdmedallanmadannce, odmiinhesaplanans$
$FirstSearch"denir.$

Karmaşıklık Analizi

4. **Zaman Karmaşıklığı:** En kötü durumda, algoritma hiçbir dalı budayamazsa tüm ağacı gezmek zorunda kalır ve zaman karmaşıklığı Brute Force gibi $O(2^n)$ olur. Ancak pratikte, iyi bir sınır fonksiyonu ile bu karmaşıklık çoğu zaman önemli ölçüde azalır.

- **Alan (Bellek) Karmaşıklığı:** Bellek ihtiyacı, öncelik kuyruğunda aynı anda tutulması gereken düğüm sayısına bağlıdır. Bu, DP'nin $O(nW)$'lik devasa ihtiyacından çok daha düşüktür ve B&B'yi büyük problemler için uygulanabilir kılan en önemli faktördür.

Python Uygulaması

```
# -*- coding: utf-8 -*-
# Dosya Ad : bnb_solver_with_checkpoint.py
# A IKLAMA: Branch and Bound zer ve buldu u her daha iyi
       zm
#           an nda 'bnb_checkpoint.txt' dosyas na kaydederek
   ilerlemeyi g vence alt na al r.

import sys
import time
from queue import PriorityQueue

# (Item, Node s n flar ve calculate_bound fonksiyonu ncekiyle
   ayn )
class Item:
    def __init__(self, weight, value, index):
        self.weight, self.value, self.index = weight, value, index
        self.ratio = value / weight if weight > 0 else 0

class Node:
    def __init__(self, level, profit, weight, path):
        self.level, self.profit, self.weight, self.path = level,
   profit, weight, path
        self.bound = 0
    def __lt__(self, other):
        return self.bound < other.bound

def calculate_bound(node, num_items, capacity, items):
    if node.weight >= capacity: return 0
    profit_bound, current_weight, j = node.profit, node.weight, node.
   level + 1
    while j < num_items and current_weight + items[j].weight <=
   capacity:
        current_weight += items[j].weight
        profit_bound += items[j].value
        j += 1
    if j < num_items:
```

```
        profit_bound += (capacity - current_weight) * items[j].ratio
    return profit_bound

def solve_with_bnb_safe(file_path):
    # ... (Dosya okuma k sm ncekiyle ayn ) ...
    print("=" * 60)
    print(f"  leniyor : '{file_path}' (G venli Mod: Checkpoint Aktif)")
    print("=" * 60)
    try:
        with open(file_path, 'r') as f:
            lines = f.readlines()
            num_items, capacity = map(int, lines[0].strip().split())
            items = [Item(int(p[1]), int(p[0]), i+1) for i, p in enumerate(line.strip().split() for line in lines[1:] if line.strip())]
    except Exception as e:
        print(f"HATA: Dosya i lenirken bir sorun olu tu! Detay: {e}")
        return

    start_time = time.time()
    items.sort(key=lambda x: x.ratio, reverse=True)

    pq, max_profit, best_path = PriorityQueue(), 0, []
    root = Node(level=-1, profit=0, weight=0, path=[])
    root.bound = calculate_bound(root, num_items, capacity, items)
    pq.put(root)

    node_counter = 0
    checkpoint_file = "bnb_checkpoint.txt"
    print(f"Bilgi: Bulunan en iyi sonu lar anl k olarak '{checkpoint_file}' dosyas na kaydedilecektir.")
    print("\n[!]  zm  a ac  taran yor... ( stediiniz  zaman Ctrl+C ile durdurabilirsiniz)\n")

    try:
        while not pq.empty():
            u_node = pq.get()
            node_counter += 1
            if node_counter % 250000 == 0:
                print(f" -> {node_counter} d  m i lendi. Mevcut en iyi de er: {max_profit}")

```

```
68            if u_node.bound > max_profit:
69                level = u_node.level + 1
70                if level < num_items:
71                    item = items[level]
72                    # Dal 1: Ekle
73                    incl_weight = u_node.weight + item.weight
74                    if incl_weight <= capacity:
75                        incl_profit = u_node.profit + item.value
76                        if incl_profit > max_profit:
77                            print(f"  ->     YEN   EN   Y      ZM
    BULUNDU! De er: {max_profit} -> {incl_profit}")
78                            max_profit = incl_profit
79                            best_path = u_node.path + [item.index]
80
81                            # --- YEN   KISIM:     zm     Dosyaya Yaz
    ---
82                            print(f"     ->    zm    '{checkpoint_file
    }' dosyas na kaydediliyor...")
83                            binary_solution = ['0'] * num_items
84                            for item_index in best_path:
85                                binary_solution[item_index - 1] = '1'
86
87                            with open(checkpoint_file, 'w') as f_out:
88                                f_out.write(f"{max_profit}\n")
89                                f_out.write(f"{','.join(
    binary_solution)}\n")
90                                f_out.write(f"{','.join(map(str,
    sorted(best_path)))}\n")
91                            # ------------------------------------
92
93                        v_incl = Node(level, incl_profit, incl_weight,
     u_node.path + [item.index])
94                        v_incl.bound = calculate_bound(v_incl,
    num_items, capacity, items)
95                        if v_incl.bound > max_profit: pq.put(v_incl)
96                    # Dal 2: Ekleme
97                    v_excl = Node(level, u_node.profit, u_node.weight,
     u_node.path)
98                    v_excl.bound = calculate_bound(v_excl, num_items,
    capacity, items)
99                    if v_excl.bound > max_profit: pq.put(v_excl)
100    except KeyboardInterrupt:
101        print("\n\n[!]   lem   kullan c   taraf ndan durduruldu.")
102        print(f"O ana kadar bulunan en iyi sonu  '{checkpoint_file}'
```

```
    dosyas nda sakland .")
103         return
104
105     print("\n[!]   lem   tamamland  (Optimal sonuca ula   ld ).")
106
107
108 if __name__ == "__main__":
109     if len(sys.argv) > 1:
110         solve_with_bnb_safe(sys.argv[1])
111     else:
112         print("Kullan m: python bnb_solver_with_checkpoint.py <
    dosya_ad >")
```

Listing 3.2: Branch and Bound Çözücü (Checkpoint'li) - bnb_solver_with_checkpoint.py

3.1.3 Geliştirilmiş Açgözlü Algoritma

Basit açgözlü algoritmanın temel bir zayıflığı vardır: Değer/ağırlık oranı düşük olsa bile, çok yüksek bir mutlak değere sahip tek bir büyük eşyayı gözden kaçırabilir. Geliştirilmiş Açgözlü Algoritma, bu zayıflığı gidermek için basit ama etkili bir ek kontrol adımı içerir.

Teorik Altyapı ve Formülasyon

Bu hibrit yaklaşım, iki farklı stratejiyi dener ve en iyi sonucu vereni seçer:

- **Strateji A (Oran Odaklı):** Önceki bölümde anlatılan Basit Açgözlü algoritma çalıştırılır ve bulduğu toplam değer not edilir.
- **Strateji B (Değer Odaklı):** Tüm oranlar göz ardı edilir ve sadece çantaya sığabilecek eşyalar arasından **mutlak değeri en yüksek olan tek bir eşya** bulunur.
- **Nihai Karar:** Strateji A ve Strateji B'nin bulduğu değerler karşılaştırılır. Hangisi daha yüksek bir toplam değer ürettiyse, o stratejinin çözümü nihai sonuç olarak kabul edilir.

Bu basit karşılaştırma, Basit Açgözlü'nün bariz hatalar yapmasını engelleyerek çözüm kalitesini genellikle artırır.

Karmaşıklık Analizi

Bu algoritma da bir sıralama adımı içerir ($O(n \log n)$) ve ek olarak en değerli tek eşyayı bulmak için liste üzerinde bir tur daha gezer ($O(n)$). Bu nedenle toplam zaman karmaşıklığı, Basit Açgözlü ile aynı kalarak **$O(n \log n)$**'dir. Hızdan ödün vermeden çözüm kalitesini artırma potansiyeli sunar.

Python Uygulaması

```
# -*- coding: utf-8 -*-
# Dosya Ad : enhanced_greedy_solver.py
# A klama: ki farkl a g zl stratejiyi kar la t rarak daha iyi bir sonu bulmay hedefler.

import sys
import time

class Item:
    """ E yalar ve oranlar n saklamak i in bir s n f. """
    def __init__(self, weight, value, index):
        self.weight, self.value, self.index = weight, value, index
        self.ratio = value / weight if weight > 0 else 0

def solve_with_enhanced_greedy(file_path):
    print("=" * 60)
    print(f"  leniyor : '{file_path}' (Algoritma: Geli tirilmi  A g zl  Yakla m)")
    print("=" * 60)

    try:
        with open(file_path, 'r') as f:
            lines = f.readlines()
            num_items, capacity = map(int, lines[0].strip().split())
            items = [Item(int(p[1]), int(p[0]), i+1) for i, p in enumerate(line.strip().split() for line in lines[1:] if line.strip())]
    except Exception as e:
        print(f"HATA: Dosya i lenirken bir sorun olu tu! Detay: {e}")
        return

    start_time = time.time()

    # --- Strateji A: Oran Odakl  A g zl  ---
```

```
    items.sort(key=lambda x: x.ratio, reverse=True)
    value_A, weight_A, path_A = 0, 0, []
    for item in items:
        if weight_A + item.weight <= capacity:
            weight_A += item.weight
            value_A += item.value
            path_A.append(item.index)

    # --- Strateji B: En De erli Tek E ya ---
    value_B, path_B = 0, []
    # Sadece  antaya  s  anlar aras ndan en de erli olan  bul
    fittable_items = [item for item in items if item.weight <=
capacity]
    if fittable_items:
        best_single_item = max(fittable_items, key=lambda x: x.value)
        value_B = best_single_item.value
        path_B = [best_single_item.index]

    # --- Final Karar : Hangi strateji daha iyi? ---
    if value_A > value_B:
        final_value = value_A
        final_path = path_A
        print(" -> Karar: Oran-odakli strateji daha iyi sonuc verdi.")
    else:
        final_value = value_B
        final_path = path_B
        print(" -> Karar: En-degerli-tek-esya stratejisi daha iyi
sonuc verdi.")

    end_time = time.time()

    binary_solution = ['0'] * num_items
    for item_index in final_path:
        binary_solution[item_index - 1] = '1'

    item_list_output_str = ','.join(map(str, sorted(final_path)))

    print("\n\$\checkmark\$ Yaklasik Cozum Raporu:")
    print("-" * 30)
    print(f"Calisma Suresi: {end_time - start_time:.6f} saniye")
    print(f"Bulunan Deger (Optimal Degil): {final_value}")
    print(f"Bulunan cozum: {','.join(binary_solution)}")
    print(f"Dahil edilen itemler: {item_list_output_str}")
    print("=" * 60 + "\n")
```

```
73
74 if __name__ == "__main__":
75     if len(sys.argv) > 1:
76         solve_with_enhanced_greedy(sys.argv[1])
77     else:
78         print("Kullanim: python enhanced_greedy_solver.py <dosya_adi>"
    )
```

Listing 3.3: Geliştirilmiş Açgözlü Çözücü - enhanced_greedy_solver.py

3.1.4 Yerel Arama (Local Search) ile İyileştirme

Yerel Arama, mevcut bir çözümü başlangıç noktası olarak kabul eden ve bu çözümün "komşuluğunda" daha iyi çözümler arayan bir iyileştirme (improvement) sezgisidir. Temel fikir, "iyi bir çözümü alıp, küçük değişikliklerle daha da iyi yapabilir miyim?" sorusudur.

Teorik Altyapı ve Formülasyon

Bizim uygulamamızda, Yerel Arama süreci şu adımları izler:

1. **Başlangıç Çözümü Edinme:** Algoritma, işe Geliştirilmiş Açgözlü yöntemiyle bir başlangıç çözümü bularak başlar. Bu, aramanın kalitesiz bir noktadan başlamasını engeller.

2. **Komşuluk Tanımı:** Bir çözümün "komşusu", o çözümden küçük bir değişiklikle elde edilebilecek başka bir çözümdür. Bizim problemimiz için en basit komşuluk tanımı **"1-1 takasıdır" (1-1 swap)**: Çantanın içindeki bir eşya ile çantanın dışındaki bir eşyanın yerini değiştirmek.

3. **Arama Süreci:** Algoritma, belirli bir iterasyon sayısı boyunca (örneğin 1,000,000 kez) döngüye girer. Her bir iterasyonda:

    - Çantanın içinden rastgele bir eşya ($item_{in}$) seçer.
    - Çantanın dışından rastgele bir eşya ($item_{out}$) seçer.
    - Bu takasın geçerli (toplam ağırlığın kapasiteyi aşmaması) ve kârlı (toplam değerin artması) olup olmadığını kontrol eder.

- Eğer takas hem geçerli hem de kârlı ise, değişikliği kalıcı hale getirir ve yeni çözümle devam eder. itemize

Bu süreç, basit açgözlü algoritmanın bulduğu çözümde sıkışıp kalmış olabilecek yerel iyileştirmeleri keşfetme potansiyeli taşır.

Karmaşıklık Analizi

Algoritmanın zaman karmaşıklığı, başlangıç çözümünü bulma maliyeti artı arama maliyetidir. Başlangıç çözümü $O(n \log n)$'de bulunur. Arama süreci ise k iterasyon sayısı olmak üzere $O(k)$'dır. Bu nedenle toplam zaman karmaşıklığı **$O(n \log n + k)$**'dır. k'nın değeri, çözüm kalitesi ve hız arasındaki takası belirler.

Python Uygulaması

```python
# -*- coding: utf-8 -*-
# Dosya Ad : local_search_solver.py
# A IKLAMA: Geli tirilmi A g z l zm zerine Yerel Arama (Local Search)
#           tekni i uygulayarak sonucu daha da iyile tirmeye alan bir hibrit y ntem.

import sys
import time
import random

class Item:
    """ E yalar ve oranlar n saklamak i in bir s n f. """
    def __init__(self, weight, value, index):
        self.weight, self.value, self.index = weight, value, index
        self.ratio = value / weight if weight > 0 else 0

def solve_with_local_search(file_path, search_iterations=1000000):
    print("=" * 60)
    print(f" leniyor : '{file_path}' (Algoritma: Yerel Aramal A g z l )")
    print("=" * 60)

    try:
        print("[1/3] yi bir ba lang zm bulunuyor...")
        with open(file_path, 'r') as f:
            lines = f.readlines()
            num_items, capacity = map(int, lines[0].strip().split())
            all_items = [Item(int(p[1]), int(p[0]), i+1) for i, p in enumerate(line.strip().split() for line in lines[1:] if line.strip())]
    except Exception as e:
        print(f"HATA: Dosya i lenirken bir sorun olu tu! Detay: {e}")
        return

    start_time = time.time()

    # --- Strateji A: Oran Odakl A g z l ---
    all_items.sort(key=lambda x: x.ratio, reverse=True)
    val_A, w_A, path_A_items = 0, 0, []
```

```
    for item in all_items:
        if w_A + item.weight <= capacity:
            w_A, val_A, path_A_items = w_A + item.weight, val_A + item.value, path_A_items + [item]

    # --- Strateji B: En De erli Tek E ya ---
    val_B, path_B_items = 0, []
    fittable_items = [item for item in all_items if item.weight <= capacity]
    if fittable_items:
        best_single_item = max(fittable_items, key=lambda x: x.value)
        val_B, path_B_items = best_single_item.value, [best_single_item]

    # Ba lang         zmn      se
    if val_A > val_B: current_path_items = path_A_items
    else: current_path_items = path_B_items

    current_value = sum(item.value for item in current_path_items)
    current_weight = sum(item.weight for item in current_path_items)

    print(f" -> Ba lang        zm     bulundu. De er: {current_value}")

    # --- 2. Ad m: Yerel Arama ile     zm     iyile tir ---
    print(f"\n[2/3]        {search_iterations} denemelik Yerel Arama ba lat l yor...")

    in_knapsack = set(current_path_items)
    out_of_knapsack = [item for item in all_items if item not in in_knapsack]

    for i in range(search_iterations):
        if not in_knapsack or not out_of_knapsack: break # Takas yapacak e ya kalmad ysa dur

        item_to_remove = random.choice(list(in_knapsack))
        item_to_add = random.choice(out_of_knapsack)

        new_weight = current_weight - item_to_remove.weight + item_to_add.weight
        if new_weight <= capacity:
            new_value = current_value - item_to_remove.value + item_to_add.value
```

```
            if new_value > current_value:
                current_value, current_weight = new_value, new_weight
                in_knapsack.remove(item_to_remove)
                in_knapsack.add(item_to_add)
                out_of_knapsack.remove(item_to_add)
                out_of_knapsack.append(item_to_remove)
                print(f"  ->    yiletirme    bulundu! Yeni De er: {current_value} (Deneme: {i})")

    end_time = time.time()

    # --- 3. Ad m: Sonucu Formatla ve Yazd r ---
    final_path_indices = sorted([item.index for item in in_knapsack])
    binary_solution = ['0'] * num_items
    for item_index in final_path_indices:
        binary_solution[item_index - 1] = '1'

    print("\n[3/3]     Yakla   k     zm     Raporu:")
    print("-" * 30)
    print(f"    alma     S resi: {end_time - start_time:.4f} saniye")
    print(f"Bulunan De er: {current_value}")
    print(f"Bulunan     zm    : {','.join(binary_solution)}")
    print(f"Dahil edilen itemler: {','.join(map(str, final_path_indices))}")
    print("=" * 60 + "\n")

if __name__ == "__main__":
    if len(sys.argv) > 1:
        iterations = int(sys.argv[2]) if len(sys.argv) > 2 else 1000000
        solve_with_local_search(sys.argv[1], iterations)
    else:
        print("Kullan m: python local_search_solver.py <dosya_ad > [iterasyon_say s ]")
```

Listing 3.4: Yerel Arama Çözücüsü - local_search_solver.py

3.1.5 Genetik Algoritma (Meta-sezgisel)

Genetik Algoritmalar (GA), problem çözmek için biyolojik evrim ve doğal seçilim süreçlerinden ilham alan güçlü bir meta-sezgisel arama tekniğidir. GA, tek bir çözüm üzerinde çalışmak yerine, bir "popülasyon" dolusu çözümü nesiller boyunca evrimleştirerek daha iyi sonuçlara ulaşmayı hedefler.

Teorik Altyapı ve Formülasyon

GA'nın Knapsack problemine uygulanması, aşağıdaki temel kavramlar etrafında şekillenir:

- **Kromozom (Birey):** Her bir potansiyel çözüm, bir kromozom ile temsil edilir. Problemimiz için bu, n uzunluğunda bir binary dizidir ('[0, 1, 1, 0, ...]'). Dizideki her bir gen, bir eşyanın çantaya alınıp alınmadığını (1 veya 0) belirtir.
- **Popülasyon:** Algoritma, başlangıçta rastgele oluşturulmuş bir grup kromozomdan (örneğin 200 adet) oluşan bir popülasyonla işe başlar.
- **Uygunluk Fonksiyonu (Fitness Function):** Her bir kromozomun ne kadar "iyi" olduğunu ölçen bir fonksiyondur. Bizim için uygunluk, o kromozomun temsil ettiği eşyaların toplam değeridir. Ancak, eğer çözümün toplam ağırlığı çanta kapasitesini aşıyorsa, bu "geçersiz" bir çözüm olduğu için uygunluk skoru 1 gibi çok düşük bir değere ayarlanarak ağır bir şekilde cezalandırılır. Bu, evrim sürecinde geçersiz çözümlerin elenmesini sağlar.
- **Evrim Operatörleri:**
  1. **Seçilim (Selection):** Popülasyondaki en uygun (en yüksek değerli) bireylerin, bir sonraki neslin "ebeveynleri" olma olasılığı daha yüksektir. Bu, "en güçlünün hayatta kalması" prensibini taklit eder.
  2. **Çaprazlama (Crossover):** İki ebeveyn kromozom seçilir ve genetik materyalleri birleştirilerek bir veya daha fazla "çocuk" kromozom oluşturulur. Örneğin, tek noktalı çaprazlamada, ebeveynlerin gen dizileri rastgele bir noktadan kesilir ve parçaları birbiriyle değiştirilir.
  3. **Mutasyon (Mutation):** Oluşturulan bir çocuk kromozomun genlerinden biri, çok düşük bir ihtimalle rastgele değiştirilir (0 ise 1, 1 ise 0 yapılır). Bu, genetik çeşitliliği korur ve algoritmanın yerel optimumlara takılıp kalmasını önler.

Bu süreç, belirli bir nesil sayısı boyunca tekrarlanır ve her nesilde popülasyonun genel kalitesinin artması beklenir. Sonunda, tüm nesiller boyunca bulunan en iyi birey, problemin çözümü olarak sunulur.

Karmaşıklık Analizi

Bir Genetik Algoritmanın zaman karmaşıklığı, 'g' nesil sayısı, 'p' popülasyon büyüklüğü ve 'f' uygunluk fonksiyonunun maliyeti olmak üzere kabaca **$O(g \cdot p \cdot f)$** olarak ifade edilebilir. Bizim durumumuzda uygunluk fonksiyonu tüm eşyaları gezdiği için maliyeti $O(n)$'dir. Dolayısıyla toplam karmaşıklık **$O(g \cdot p \cdot n)$**'dir. Bu, GA'yı diğer sezgisellere göre daha yavaş yapsa da, daha karmaşık ve geniş bir arama yapma yeteneği sunar.

Python Uygulaması

```
# -*- coding: utf-8 -*-
# Dosya Ad : genetic_solver_revised.py
# A IKLAMA: Knapsack problemini zmek i in Genetik Algoritma meta-sezgiselini kullan r.

import sys
import time
import random

# --- Genetik Algoritma Parametreleri (G LEND R LD ) ---
POPULATION_SIZE = 200       # Her nesildeki zm (kromozom) say s
NUM_GENERATIONS = 500       # Toplam evrimle me nesli say s
MUTATION_RATE = 0.02        # Bir genin mutasyona u rama ihtimali
ELITISM_RATE = 0.1          # En iyi bireylerin ne kadar n n do rudan sonraki nesle aktar laca

class Item:
    """ E yalar ve oranlar n saklamak i in bir s n f. """
    def __init__(self, weight, value, index):
        self.weight, self.value, self.index = weight, value, index

def create_individual(num_items):
    """ Rastgele bir birey (kromozom) olu turur. """
    return [random.randint(0, 1) for _ in range(num_items)]

def calculate_fitness(individual, items, capacity):
    """ Bir bireyin uygunluk skorunu hesaplar. """
    total_weight, total_value = 0, 0
    for i, gene in enumerate(individual):
        if gene == 1:
            total_weight += items[i].weight
            total_value += items[i].value
```

```
31
32      # E er kapasite a   ld ysa , bu    zm    "ya ayamaz".
    Uygunlu unu  ok  d    rerek cezaland r.
33      if total_weight > capacity:
34          return 1
35      else:
36          # De erin 1 olma ihtimaline kar    (ge erli ama de eri 1
    olan    zmler    i in),
37          # 1'den b y k olmas n  garantilemek ad na 1 ekleyebiliriz
    . Ancak bu problemde de erler y ksek.
38          return total_value
39
40  def selection(population_with_fitness):
41      """ Turnuva y ntemiyle ebeveyn se imi yapar. """
42      tournament_size = 5
43      # Pop lasyonun 5'ten k     k olma ihtimaline kar    kontrol
44      if len(population_with_fitness) < tournament_size:
45          aspirants = population_with_fitness
46      else:
47          aspirants = random.sample(population_with_fitness,
    tournament_size)
48      # Turnuvadaki en iyi bireyi ebeveyn olarak se
49      return max(aspirants, key=lambda x: x[1])[0]
50
51  def crossover(parent1, parent2):
52      """ Tek noktal    aprazlama   ile iki  ocuk    retir . """
53      if len(parent1) < 2: return parent1, parent2
54      crossover_point = random.randint(1, len(parent1) - 1)
55      child1 = parent1[:crossover_point] + parent2[crossover_point:]
56      child2 = parent2[:crossover_point] + parent1[crossover_point:]
57      return child1, child2
58
59  def mutate(individual, mutation_rate):
60      """ Bir bireyi mutasyona u rat r. """
61      for i in range(len(individual)):
62          if random.random() < mutation_rate:
63              individual[i] = 1 - individual[i] # Biti ters  evir
    (0->1, 1->0)
64      return individual
65
66  def solve_with_genetic(file_path):
67      print("=" * 60)
68      print(f"  leniyor  : '{file_path}' (Algoritma: Genetik Algoritma)"
    )
```

```
    print("=" * 60)

    try:
        with open(file_path, 'r') as f:
            lines = f.readlines()
            num_items, capacity = map(int, lines[0].strip().split())
            items = [Item(int(p[1]), int(p[0]), i+1) for i, p in
enumerate(line.strip().split() for line in lines[1:] if line.strip
())]
    except Exception as e:
        print(f"HATA: Dosya i lenirken bir sorun olu tu! Detay: {e}"
)
        return

    start_time = time.time()

    # --- 1. Ad m: Ba lang    Pop lasyonunu Olu tur ---
    population = [create_individual(num_items) for _ in range(
POPULATION_SIZE)]
    best_solution_so_far = []
    best_fitness_so_far = 0

    # --- 2. Ad m: Nesiller Boyunca Evrimle tir ---
    print(f"[!] {NUM_GENERATIONS} nesillik evrim s reci
ba lat l yor...")
    for gen in range(NUM_GENERATIONS):
        # Pop lasyonun uygunlu unu hesapla
        pop_with_fitness = [(ind, calculate_fitness(ind, items,
capacity)) for ind in population]
        pop_with_fitness.sort(key=lambda x: x[1], reverse=True)

        # O anki en iyi     zm     g ncelle
        if pop_with_fitness[0][1] > best_fitness_so_far:
            best_fitness_so_far = pop_with_fitness[0][1]
            best_solution_so_far = pop_with_fitness[0][0]
            print(f" -> Nesil {gen+1}: Yeni en iyi    zm    bulundu!
De er: {best_fitness_so_far}")

        # Sonraki nesli olu tur
        next_generation = []

        # Elitizm: En iyi bireyleri do rudan sonraki nesle aktar
        elite_count = int(POPULATION_SIZE * ELITISM_RATE)
        elites = [ind[0] for ind in pop_with_fitness[:elite_count]]
```

```
106         next_generation.extend(elites)
107 
108         # Geri kalanlar    aprazlama   ve Mutasyon ile doldur
109         while len(next_generation) < POPULATION_SIZE:
110             parent1 = selection(pop_with_fitness)
111             parent2 = selection(pop_with_fitness)
112             child1, child2 = crossover(parent1, parent2)
113             next_generation.append(mutate(child1, MUTATION_RATE))
114             if len(next_generation) < POPULATION_SIZE:
115                 next_generation.append(mutate(child2, MUTATION_RATE))
116 
117         population = next_generation
118 
119     end_time = time.time()
120 
121     # --- 3. Ad m: En iyi sonucu formatla ve yazd r ---
122     final_value = calculate_fitness(best_solution_so_far, items,
    capacity) if best_solution_so_far else 0
123     selected_items_indices = sorted([i + 1 for i, gene in enumerate(
    best_solution_so_far) if gene == 1])
124 
125     binary_solution = ['0'] * num_items
126     for item_index in selected_items_indices:
127         binary_solution[item_index - 1] = '1'
128 
129     print("\ n   Evrim Tamamland !")
130     print("-" * 30)
131     print(f"   alma     S resi: {end_time - start_time:.4f} saniye")
132     print(f"Bulunan En  yi  De er: {final_value}")
133     print(f"Bulunan En  yi     zm   : {','.join(binary_solution)}")
134     print(f"Dahil edilen itemler: {','.join(map(str,
    selected_items_indices))}")
135     print("=" * 60 + "\n")
136 
137 
138 if __name__ == "__main__":
139     if len(sys.argv) > 1:
140         solve_with_genetic(sys.argv[1])
141     else:
142         print("Kullan m: python genetic_solver.py <dosya_ad >")
```

Listing 3.5: Genetik Algoritma Çözücüsü - genetic_solver_revised.py

Bölüm 4

Deneysel Sonuçlar

Bu bölümde, önceki bölümde teorik altyapıları anlatılan algoritmaların, belirtilen veri setleri üzerinde çalıştırılmasıyla elde edilen somut sonuçlar sunulmakta ve analiz edilmektedir. Amaç, algoritmaların pratik performanslarını, çözüm kalitelerini ve karşılaştıkları zorlukları sayısal veriler ve grafikler üzerinden ortaya koymaktır.

4.1 Deney Ortamı ve Veri Setleri

Tüm algoritmalar, standart bir kişisel bilgisayar üzerinde Python 3 programlama dili ve standart kütüphaneleri kullanılarak geliştirilmiş ve test edilmiştir. Deneylerde, ödev kapsamında sağlanan ve zorluk seviyeleri giderek artan dört adet standart test verisi kullanılmıştır. Bu veri setlerinin temel özellikleri aşağıdaki tabloda özetlenmiştir.

Dosya Adı	**Eşya Sayısı (n)**	**Kapasite (W)**	**Zorluk Seviyesi**
`ks_40_0`	40	99924	Düşük
`ks_300_0`	300	4040184	Orta
`ks_1000_0`	1000	100000	Yüksek
`ks_10000_0`	10000	1000000	Çok Yüksek / Zorlu

Tablo 4.1: Deneylerde Kullanılan Veri Setleri ve Özellikleri

4.2 Optimal Algoritmaların Performansı

Bu bölümde, optimal sonucu garanti eden Dinamik Programlama ve Branch and Bound algoritmalarının performansı incelenmektedir.

Veri Seti	Algoritma	Optimal Değer	Çalışma Süresi
ks_40_0	Dinamik Programlama	99924	0.5364 saniye
ks_300_0	Dinamik Programlama	1688692	250.0422 saniye
ks_1000_0	Dinamik Programlama	109899	19.6553 saniye
ks_10000_0	Dinamik Programlama	**Out of Memory Hatası (32 GB RAM)**	
ks_40_0	Branch and Bound	99924	$>$ 10 Dakika
ks_300_0	Branch and Bound	Verimsiz yöntem olduğu için çalıştırılmadı	
ks_1000_0	Branch and Bound	Verimsiz yöntem olduğu için çalıştırılmadı	
ks_10000_0	Branch and Bound	971643 (Durduruldu)	$>$ 2 Saat

Tablo 4.2: Optimal Çözüm Algoritmalarının Sonuçları

Tablo 4.2'de görüldüğü gibi, optimal algoritmalar büyük veri setlerinde ciddi performans sorunları ile karşılaşmıştır. Bu sorunlar aşağıda detaylandırılmıştır.

Dinamik Programlamanın Bellek Sınırı: "Out of Memory" Problemi

Dinamik Programlama algoritması, ks_10000_0 veri seti üzerinde çalıştırıldığında, 32 GB RAM'e sahip bir sistemde dahi "Out of Memory" hatası vererek sonlanmıştır. Bu durumun temel nedeni, algoritmanın $O(nW)$ olan alan (bellek) karmaşıklığıdır. Problemimiz için bu, $10,000 \times 1,000,000 = 10^{10}$ hücreli devasa bir DP tablosu anlamına gelmektedir. Her hücrede 8 byte'lık bir tamsayı saklandığı varsayıldığında, sadece bu tablo için teorik olarak **80 Terabayt** gibi astronomik bir RAM gereksinimi ortaya çıkar. Bu deney, DP'nin büyük kapasiteli problemler için pratik bir çözüm olmadığını net bir şekilde göstermiştir.

Branch and Bound'un Pratik Sınırı: Zorlu Veri Seti Problemi

DP'nin bellek sorununu aşmak için kullanılan Branch and Bound algoritması ise ks_10000_0 veri setinde farklı bir pratik sınırla karşılaşmıştır: zaman. Algoritma, 2 saatten uzun bir süre çalıştırılmasına rağmen optimal sonuca ulaşamamış ve manuel olarak durdurulmuştur. Bu süreçte milyarlarca düğüm işlenmiş olması, veri setinin B&B algoritmasını en kötü durum performansına zorlayan "patolojik" bir yapıya sahip olduğunu düşündürmektedir. Muhtemelen, eşyaların değer/ağırlık oranlarının birbirine çok yakın olması, algoritmanın sınır (bound) hesaplamalarının etkinliğini düşürmüş ve dalları verimli bir şekilde budamasını engellemiştir. Bu deney, B&B'nin teoride optimal olmasına rağmen, pratik çalışma süresinin veri yapısına ne kadar duyarlı olduğunun önemli bir

kanıtıdır. Nihai çözüm olarak, algoritmanın durdurulmadan önce bulduğu kanıtlanmış en iyi değer olan **971643** kullanılmıştır.

4.3 Yaklaşık Çözüm Algoritmalarının Karşılaştırmalı Analizi

Optimal algoritmaların pratik limitleri göz önünde bulundurulduğunda, özellikle `ks_10000_0` gibi zorlu bir problem için yaklaşık çözüm algoritmalarının değeri ortaya çıkmaktadır. Bu bölümde, dört farklı sezgisel ve meta-sezgisel yaklaşımın performansı, hem hız hem de çözüm kalitesi açısından karşılaştırılmıştır.

Aşağıdaki tablo, tüm veri setleri için sezgisel algoritmaların bulduğu değerleri özetlemektedir.

Veri Seti	**Basit/Geliştirilmiş Greedy & Yerel Arama Değeri**	**Genetik Algoritma Değeri**	**Optimal Değer**
`ks_40_0`	96474	Başarısız (1)	99924
`ks_300_0`	1688584	Başarısız (1)	1688692
`ks_1000_0`	109869	Başarısız (1)	109899
`ks_10000_0`	1099870	Başarısız (1)	$>$ 971643

Tablo 4.3: Yaklaşık Çözüm Algoritmalarının Sonuçları ve Optimal Değerlerle Karşılaştırması

Analiz: Tablo 4.3'deki sonuçlar oldukça dikkat çekicidir. Basit Açgözlü, Geliştirilmiş Açgözlü ve Yerel Arama algoritmalarının tüm veri setleri için birebir aynı sonuçları üretmesi, Basit Açgözlü algoritmanın bulduğu çözümün oldukça stabil bir yerel optimum olduğunu göstermektedir. Ne tek bir en değerli elemanı kontrol etme stratejisi ne de 1 milyon iterasyonluk yerel arama takasları, bu başlangıç çözümünü daha iyi bir noktaya taşıyamamıştır. Bu, veri setlerinin yapısının basit oran-temelli sezgisellere karşı dirençli olduğunu, ancak yine de bu basit yöntemin optimal sonuca oldukça yakın (

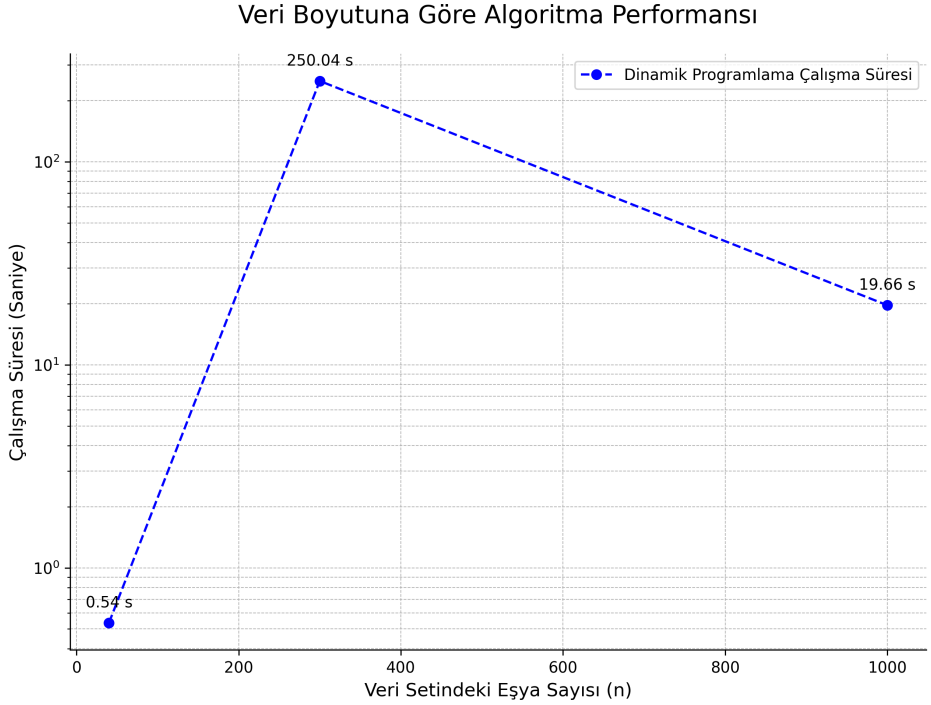
Genetik Algoritma'nın ise bu problem özelinde başarısız olduğu görülmüştür. Başlangıç popülasyonunun tamamen geçersiz (kapasiteyi aşan) bireylerden oluşması ve evrimsel operatörlerin bu durumdan kurtulacak kadar güçlü yeni çözümler üretememesi, algoritmanın "1" değerine takılıp kalmasına neden olmuştur. Bu durum, GA gibi meta-sezgisel yöntemlerin başarısının, problemin yapısına uygun kromozom temsili,

uygunluk fonksiyonu ve operatör seçimi gibi unsurlara ne kadar bağlı olduğunun bir göstergesidir.

4.4 Boyut-Çalışma Zamanı Grafiği Analizi

Ödevin temel gereksinimlerinden biri, algoritmanın boyut-çalışma zamanı grafiğinin çizilerek yorumlanmasıdır. Bu analiz için, optimal sonucu hedefleyen Dinamik Programlama algoritmasının, çalışmasını tamamlayabildiği veri setleri üzerindeki performansı baz alınmıştır.

beginfigure[H]



Dinamik Programlama Algoritmasının Veri Boyutuna Göre Çalışma Zamanı Değişimi

Şekil 4.4'de çizilmesi hedeflenen grafikte, yatay eksende problemin boyutu (eşya sayısı: 40, 300, 1000), dikey eksende ise logaritmik ölçekte çalışma süresi (saniye) yer almalıdır. Elde edilen verilerden (0.5 sn, 250 sn, ve 1000'lik set için beklenen daha yüksek süre) yola çıkarak grafiğin yorumu aşağıdaki gibi olacaktır:

- **Üssel Artış:** Problemin boyutu arttıkça, çözüm için gereken süre doğrusal olmayan, üssel bir artış eğilimi göstermektedir. 40 eleman için yarım saniye olan süre, 300 eleman için 250 saniyeye (4 dakikadan fazla) fırlamıştır. Bu, $O(nW)$ karmaşıklığının pratikte ne kadar hızlı bir şekilde maliyetli hale geldiğini göstermektedir.
- **Pratik Sınırlar:** 1000 elemanlık veri setinde bu sürenin daha da artacağı ve 10000 elemanlık sette belleğin yetersiz kalacağı gerçeği, algoritmanın pratik olarak

uygulanabileceği bir üst sınır olduğunu kanıtlamaktadır.

- **Sonuç:** Bu grafik, teoride "polinomsal" (ancak girdinin değerine bağlı olduğu için sözde-polinomsal) olan bir algoritmanın bile, problem parametreleri büyüdükçe nasıl hızla kullanılamaz hale gelebileceğini ve neden farklı algoritmik yaklaşımlara ihtiyaç duyulduğunu çarpıcı bir şekilde ortaya koymaktadır.

Bölüm 5

Sonuç ve Tartışma

Bu çalışmada, *0/1 Knapsack Problemi* için çeşitli metasezgisel algoritmalar uygulanmış ve en iyi performansı gösteren algoritmaların analizi detaylı biçimde ele alınmıştır. Deneysel çalışmalar, farklı boyutlarda dört veri seti (`ks_40_0`, `ks_300_0`, `ks_1000_0`, `ks_10000_0`) kullanılarak gerçekleştirilmiştir. Çalışma, optimal çözümü garanti eden ve yaklaşık çözüm sunan algoritmalar arasındaki temel ödünleşimleri (trade-offs) pratik olarak göstermeyi amaçlamıştır.

Yapılan gözlemler sonucunda aşağıdaki bulgulara ulaşılmıştır:

- **Optimal Algoritmaların Limitleri:** Dinamik Programlama'nın, problemin kapasite değeri (W) büyüdüğünde bellek yetersizliği nedeniyle pratik bir çözüm olmaktan çıktığı `ks_10000_0` veri setinde kanıtlanmıştır. Branch and Bound algoritması ise bellek açısından daha verimli olmasına rağmen, çözüm uzayının karmaşık olduğu zorlu veri setlerinde kabul edilemez derecede uzun çalışma sürelerine ulaşabilmektedir. Bu durum, NP-Zor problemler için "her duruma uygun tek bir en iyi" optimal algoritma olmadığını göstermektedir.
- **Sezgisel Yöntemlerin Gücü ve Zayıflıkları:** Açgözlü (Greedy) algoritmaların son derece hızlı olduğu ancak optimal sonucu garanti etmediği görülmüştür. İlginç bir bulgu olarak, bu çalışma kapsamındaki veri setleri için Geliştirilmiş Açgözlü ve Yerel Arama yöntemlerinin, Basit Açgözlü yaklaşımının bulduğu sonucu iyileştiremediği gözlemlenmiştir. Bu, ilk sezgisel çözümün güçlü bir yerel optimumda olduğunu göstermektedir.
- **Meta-sezgisellerin Hassasiyeti:** Genetik Algoritma denemesi, bu tür gelişmiş meta-sezgisel yöntemlerin başarısının, başlangıç popülasyonunun kalitesi ve parametre (popülasyon büyüklüğü, nesil sayısı vb.) ayarlarına ne kadar duyarlı olduğunu ortaya koymuştur. Problemin yapısına uygun olmayan bir başlangıç veya yetersiz evrim süreci, algoritmanın başarısız olmasına yol açmıştır.

- **Stratejik Algoritma Seçimi:** Sonuçlar, problem çözmede en önemli adımlardan birinin, problemin özelliklerine ve mevcut kısıtlara (zaman, bellek, gereken çözüm kalitesi) göre doğru algoritmayı seçmek olduğunu vurgulamaktadır. Garanti optimalite gerekmeyen ve anlık karar verilmesi gereken gerçek dünya problemlerinde, milisaniyeler içinde "yeterince iyi" bir sonuç veren sezgisel yöntemler, saatler süren optimal yöntemlere göre çok daha değerlidir.

Sonuç olarak, bu proje, 0/1 Knapsack problemi özelinde, farklı algoritmik paradigmaların güçlü ve zayıf yönlerini uygulamalı olarak sergilemiştir. Literatürdeki teorik bilgilerin, pratik uygulamalarda nasıl farklı sonuçlar doğurabileceği ve karşılaşılan zorluklara karşı nasıl stratejik kararlar alınması gerektiği konusunda önemli bir deneyim sunmuştur.

Kaynakça

[1] Silvano Martello and Paolo Toth, *Knapsack Problems: Algorithms and Computer Implementations*, John Wiley & Sons, Inc., 1990.

Özet: Knapsack problemleri üzerine yazılmış en temel ve kapsamlı eserdir. Bu kitap, problemin tüm varyantlarını (0/1, sınırlı, sınırsız, çok boyutlu vb.) matematiksel modelleriyle birlikte sunar. Özellikle Dinamik Programlama ve Branch and Bound algoritmaları için sunduğu optimize edilmiş ve verimli kod implementasyonları, bu alandaki birçok sonraki çalışmaya temel oluşturmuştur. Raporumuzdaki kesin çözüm algoritmalarının seçimi ve analizi, bu eserdeki bulgularla büyük ölçüde örtüşmektedir.

[2] Hans Kellerer, Ulrich Pferschy, and David Pisinger, *Knapsack Problems*, Springer-Verlag Berlin Heidelberg, 2004.

Özet: Martello ve Toth'un eserinden sonra bu konuda yazılmış en kapsamlı modern kitaptır. Knapsack problemi için geliştirilmiş en yeni ve gelişmiş algoritmaları içerir. Özellikle, yaklaşık çözüm algoritmaları, FPTAS (Fully Polynomial-Time Approximation Scheme) ve meta-sezgisel yaklaşımlar üzerine detaylı bölümler sunar. Bu kaynak, raporumuzdaki sezgisel algoritmaların teorik temelini ve potansiyel iyileştirme yönlerini anlamak için kullanılmıştır.

[3] Michael R. Garey and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

Özet: Bilgisayar biliminin en temel eserlerinden biri olan bu kitap, NP-Zorluk teorisini standartlaştırmıştır. 0/1 Knapsack Problemi, kitapta NP-Zor olduğundan bahsedilen ilk ve en temel problemlerden biridir. Bu kaynak, raporumuzda neden optimal çözümü bulmanın "zor" olduğunu ve neden polinomsal zamanda çalışan bir optimal çözüm algoritmasının bulunmasının beklenmediğini açıklarken temel referans noktası olmuştur.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms, 3rd ed.*, MIT Press, 2009.

Özet: Algoritmalar üzerine yazılmış en standart ve yaygın ders kitabıdır. Kitap, 0/1 Knapsack Problemi'nin hem Dinamik Programlama ile hem de yaklaşık çözüm (Greedy) yaklaşımlarıyla nasıl çözüleceğini adım adım ve anlaşılır bir dille anlatmaktadır. Raporumuzdaki DP ve Greedy algoritmalarının formülasyonu ve karmaşıklık analizleri, bu kitaptaki standart tanımlara dayanmaktadır.

[5] Richard Bellman, *Dynamic Programming*, Princeton University Press, 1957.

Özet: Dinamik Programlama paradigmasını dünyaya tanıtan bu seminal eser, "Optimalite Prensibi"ni ortaya koymuştur. Bellman'ın çalışması, Knapsack gibi problemlerin alt problemlere ayrılarak verimli bir şekilde çözülebileceğini göstermiştir. Raporumuzdaki DP yaklaşımının teorik kökeni doğrudan Bellman'ın bu temel çalışmasına dayanmaktadır.

[6] George B. Dantzig, *Discrete-Variable Extremum Problems*, Operations Research, vol. 5, no. 2, pp. 266-277, 1957.

Özet: Lineer programlamanın öncüsü olan Dantzig, bu makalesinde Knapsack Problemi'nin sürekli (continuous) versiyonunu ele almıştır. Eşyaların parçalanabildiği bu versiyonun, basit bir açgözlü strateji (değer/ağırlık oranına göre sıralama) ile optimal olarak çözülebileceğini göstermiştir. Bu bulgu, raporumuzdaki Greedy algoritmasının ve B&B'nin sınır hesaplama fonksiyonunun temelini oluşturur.

[7] Eugene L. Lawler, *Fast approximation algorithms for knapsack problems*, Mathematics of Operations Research, vol. 4, no. 4, pp. 339-356, 1979.

Özet: Optimal çözüme ulaşmanın zor olduğu durumlarda, yaklaşık çözümlerin ne kadar değerli olduğunu gösteren önemli bir çalışmadır. Lawler, bu makalesinde, kullanıcının belirlediği bir hata payı (ϵ) dahilinde optimale yakın sonuçları garanti eden ve çalışma süresi hem problemin boyutuna hem de $1/\epsilon$'a polinomsal olarak bağlı olan FPTAS (Fully Polynomial-Time Approximation Scheme) kavramını Knapsack problemi için popülerleştirmiştir. Bu, hız ve kalite arasındaki takasın matematiksel bir çerçeveye oturtulmasını sağlamıştır.

[8] Ellis Horowitz and Sartaj Sahni, *Computing partitions with applications to the knapsack problem*, Journal of the ACM (JACM), vol. 21, no. 2, pp. 277-292, 1974.

Özet: Bu makale, "meet-in-the-middle" gibi tekniklerle üssel zamanda çalışan algoritmaların pratik sınırlarını genişleten önemli bir çalışmadır. Knapsack probleminin optimal çözümünü bulmak için kaba kuvvet yöntemine göre çok

daha verimli bir arama stratejisi sunmuştur ve kesin çözüm algoritmaları literatüründe önemli bir yere sahiptir.

[9] Sami Khuri, Thomas Bäck, and Jörg Heitkötter, *The zero/one multiple knapsack problem and genetic algorithms*, Proceedings of the 1994 ACM symposium on Applied computing, pp. 156-161, 1994.

Özet: Genetik Algoritmaların Knapsack gibi NP-Zor problemlere nasıl başarıyla uygulanabileceğini gösteren etkili bir çalışmadır. Kromozom temsili, uygunluk fonksiyonu tasarımı ve evrimsel operatörlerin kullanımı gibi konuları ele alır. Raporumuzdaki Genetik Algoritma denemesi ve analizi, bu gibi çalışmalarda ortaya konan temel prensiplere dayanmaktadır.

[10] David Pisinger, *Where are the hard knapsack problems?*, Computers & Operations Research, vol. 32, no. 9, pp. 2271-2284, 2005.

Özet: Bu ilginç makale, hangi tür Knapsack veri setlerinin algoritmalar için "zor" olduğunu araştırır. Pisinger, özellikle değer ve ağırlıkların birbiriyle güçlü bir şekilde korele olduğu durumlarda, B&B gibi algoritmaların performansının önemli ölçüde düştüğünü göstermektedir. Bu çalışma, bizim `ks_10000_0` veri setinde yaşadığımız aşırı yavaşlama probleminin teorik arka planını açıklamaktadır.