BURSA ULUDAĞ ÜNİVERSİTESİ

BİLGİSAYAR MÜHENDİSLİĞİ

2024-2025 EĞİTİM ÖĞRETİM YILI GÜZ DÖNEMİ

ALGORİTMA ANALİZİ RAPORU

İleri Seviye

Dijkstra ve Contraction Hierarchies Karşılaştırması

032290004 BARIŞ IŞIK

032290004@ogr.uludag.edu.tr

032290008 MURAT BERK YETİŞTİRİR

032290008@ogr.uludag.edu.tr

032290037 BUĞRA ÖZGEN

032290037@ogr.uludag.edu.tr

1.0 Kodlama ve Simülasyon

Bu bizim simülasyonu yapacağımız ortam. CH ve Dijkstra için simülasyon sonuçlarını detaylı inceleyeceğiz.

1.1 Dijkstra Algoritması

dijkstra(): Bu, tek bir başlangıç düğümünden grafikteki diğer tüm düğümlere en kısa yolu bulmak için kullanılan bir algoritmadır. Algoritmanın temel çalışma prensibi, başlangıç düğümünden başlayarak, grafikteki her düğüme olan en kısa mesafeyi iteratif olarak güncellemektir.

- Öncelikle, başlangıç düğümüne olan mesafe sıfır olarak ayarlanır ve diğer tüm düğümlerin mesafesi sonsuz olarak belirlenir.
- Bir **öncelik kuyruğu (priority queue)** kullanılarak, her iterasyonda en düşük maliyetli düğüm seçilir. Bu, algoritmanın her zaman en kısa yolda ilerlemesini sağlar.
- Seçilen düğümün komşuları ziyaret edilir ve her bir komşunun mevcut mesafesi güncellenir. Eğer daha kısa bir yol bulunursa, bu düğümün mesafesi güncellenir ve tekrar öncelik kuyruğuna eklenir.
  Bu algoritma, özellikle yoğun grafikler için verimlidir ve ağırlıklı kenarlara sahip yönlendirilmiş ya da yönlendirilmemiş grafiklerde kullanılabilir.

```python
def dijkstra(self, start_vertex):
    vertices = list(self.G.nodes())
    visited = set()
    parents = {}
    adj = {v: [] for v in vertices}
    exist = {v: 0 for v in vertices}
    D = {v: float('inf') for v in vertices}

    parents[start_vertex] = start_vertex
    D[start_vertex] = 0
    exist[start_vertex] = 1
    print("asd")
    pq = [(0, start_vertex)]
    while pq:
        current_cost, current_vertex = heapq.heappop(pq)
        if current_vertex in visited:
            continue
        visited.add(current_vertex)

        current_neighbors = list(self.G.neighbors(current_vertex))
        for neighbor in current_neighbors:
            old_cost = D[neighbor]
            new_cost = D[current_vertex] + self.times_all[current_vertex][neighbor]
            if new_cost < old_cost:
                parents[neighbor] = current_vertex
                exist[neighbor] = exist[current_vertex]
                if neighbor not in adj[current_vertex]:
                    adj[current_vertex].append(neighbor)
                D[neighbor] = new_cost
                heapq.heappush(pq, (new_cost, neighbor))

    self.shortest_adj[start_vertex] = adj
    self.shortest_cnt[start_vertex] = exist
    print("dijkstra end")
    return D, parents
```

dijkstra_all_pairs(): Bu yöntem, grafikteki her bir düğüm çifti arasındaki en kısa yolları hesaplamak için klasik Dijkstra algoritmasını tekrar tekrar çalıştırır.

- İlk adımda, grafikteki tüm düğümler başlangıç düğümü olarak seçilir. Her bir düğüm için Dijkstra algoritması çalıştırılır ve bu düğümden diğer tüm düğümlere olan en kısa yollar hesaplanır.
- Çıktı olarak, düğüm çiftlerinin en kısa yollarını içeren bir tablo veya matris elde edilir. Bu yöntem, özellikle düğüm çiftlerinin birbirine olan mesafelerinin sık sık sorgulandığı durumlar için uygundur.

Ancak bu yöntemin karmaşıklığı, düğüm sayısının artmasıyla oldukça büyüdüğü için büyük grafikleri işlerken daha verimli yöntemlere ihtiyaç duyulabilir.

```python
def dijkstra_all_pairs(self):
    vertices = list(self.G.nodes())
    cnt =  1
    t1 = time.time()
    for u in vertices:
        if cnt == 16:
            break
        print("HEREE")
        print(cnt)
        cnt += 1
        for v in vertices:
            dist, parents = self.dijkstra(u)
            p = self.get_path(v, parents)
    t2 = time.time()
    print("HERE")
    print(t2 - t1, (t2 - t1) / (15*4397))
```

get_shortest_path_dijkstra(): Bu yöntem, belirli bir **başlangıç düğümünden** belirli bir **hedef düğüme** kadar olan en kısa yolu bulmak için Dijkstra algoritmasını özelleştirir.

- Dijkstra algoritmasının temel mantığını kullanarak, başlangıç düğümünden hedef düğüme kadar olan en kısa yolu ve bu yolun toplam maliyetini hesaplar.
- Yol, başlangıç düğümünden başlayarak hedef düğüme kadar olan tüm ara düğümleri içerir.
- Özellikle navigasyon sistemlerinde veya belirli kaynak-hedef ilişkilerinde sıklıkla kullanılır.

```
def get_shortest_path_dijkstra(self, start_stop, end_stop):
    shortest_time, parents = self.dijkstra(start_stop)
    print("ASD")
    path = self.get_path(end_stop, parents)
    if not path:
        print(f'No path found from {start_stop} to {end_stop}!')
        return 0, path
    return shortest_time[end_stop], path
```

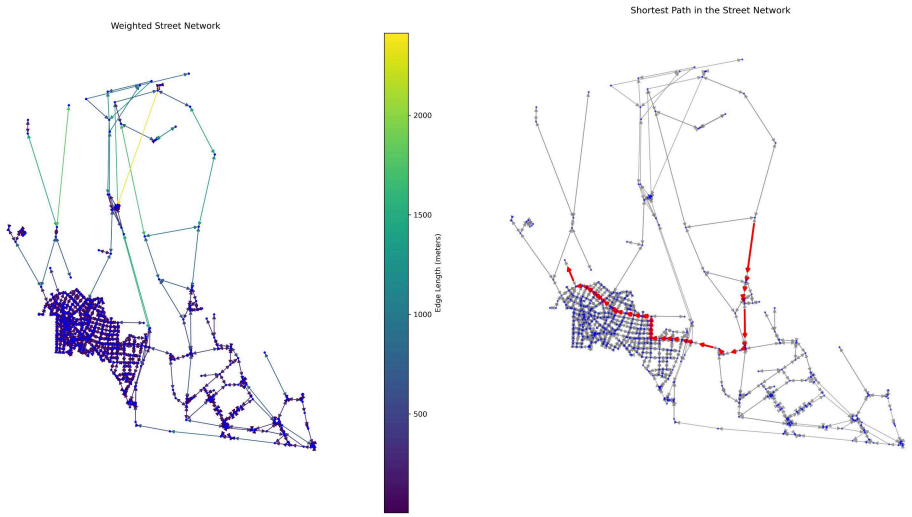
bidirectional_dijkstra(): Bu, Dijkstra algoritmasının optimize edilmiş bir versiyonudur ve büyük grafiklerde çok daha hızlı sonuçlar üretebilir.

- Algoritma, aynı anda iki Dijkstra algoritması çalıştırır: biri kaynak düğümden başlar, diğeri hedef düğümden başlar.
- Her iki algoritma da ortada buluşana kadar genişlemeye devam eder.
- Bu yaklaşımın temel avantajı, bir yerine iki arama yönünün kullanılmasının toplam arama alanını önemli ölçüde küçültmesidir. Özellikle düğüm sayısının çok fazla olduğu büyük grafiklerde bu yöntem oldukça etkilidir.
- Algoritmanın çıktısı, her iki aramanın buluştuğu noktaları dikkate alarak en kısa yolu hesaplar.

```python
def bidirectional_dijkstra(self, source_node, target_node):
    vertices = list(self.G.nodes())
    visited_start = set()
    visited_end = set()
    parents1 = {}
    parents2 = {}
    dist1 = {v: float('inf') for v in vertices}
    dist2 = {v: float('inf') for v in vertices}

    parents1[source_node] = source_node
    parents2[target_node] = target_node
    dist1[source_node] = 0
    dist2[target_node] = 0
    pq_start = [(0, source_node)]
    pq_end = [(0, target_node)]
    while pq_start or pq_end:
        if pq_start:
            _, current_vertex = heapq.heappop(pq_start)
            if current_vertex in visited_start:
                continue
            visited_start.add(current_vertex)

            for neighbor in self.G.neighbors(current_vertex):
                if self.order_of[neighbor] <= self.order_of[current_vertex]:
                    continue

                new_cost = dist1[current_vertex] + self.times_all[current_vertex][neighbor]
                if new_cost < dist1[neighbor]:
                    parents1[neighbor] = current_vertex
                    dist1[neighbor] = new_cost
                    heapq.heappush(pq_start, (new_cost, neighbor))
```

```python
        if pq_end:
            _, current_vertex = heapq.heappop(pq_end)
            if current_vertex in visited_end:
                continue
            visited_end.add(current_vertex)

            for neighbor in self.G.predecessors(current_vertex):
                if self.order_of[neighbor] <= self.order_of[current_vertex]:
                    continue

                new_cost = dist2[current_vertex] + self.times_all[neighbor][current_vertex]
                if new_cost < dist2[neighbor]:
                    parents2[neighbor] = current_vertex
                    dist2[neighbor] = new_cost
                    heapq.heappush(pq_end, (new_cost, neighbor))


    L = [v for v in self.G.nodes if dist1[v] != float('inf') and dist2[v] != float('inf')]
    if not L:
        return 0, []

    shortest_time = math.inf
    common_node = 0
    for v in L:
        if shortest_time > dist1[v] + dist2[v]:
            shortest_time = dist1[v] + dist2[v]
            common_node = v
```

1.1.1 Sonuç



Bu haritada Dijkstra algoritmasını kullanarak, okuldan Görükle'ye giden en kısa rotayı nasıl bulabileceğimizi göstereceğim. Başlangıç noktamız okul ve hedefimiz Görükle'deki bir belirli lokasyon, örneğin Görükle meydanı. Haritada okul, Görükle, ve aradaki diğer konumlar (duraklar, kavşaklar gibi) düğümler olarak temsil ediliyor ve bu düğümleri birbirine bağlayan yollar ise kenarlar olarak gösteriliyor. Her bir kenarın üzerinde, iki konum arasındaki mesafeyi veya tahmini seyahat süresini belirten bir ağırlık bulunuyor.

Dijkstra algoritması, okuldan başlayarak her bir komşu düğüm için mesafeleri hesaplar ve en kısa mesafeye sahip olan düğümü genişletmeye devam eder. Örneğin, okuldan öncelikle en yakın kavşağa, ardından bir sonraki duraklara ilerleyerek, toplam mesafeyi en aza indiren bir yol ağacı oluşturur. Bu süreç sonunda, okuldan Görükle'ye gitmek için izlenmesi gereken en kısa ve en hızlı rotayı belirleriz. Bu yöntem, günlük hayatta en kısa veya en hızlı rotayı bulmak için oldukça etkili bir çözüm sunar.

1.2 Contraction Hierarchies (CH)

preprocess(): Contraction Hierarchies, büyük grafiklerde kısa yol sorgularını hızlandırmak için kullanılan bir yöntemdir ve bu yöntem önceden yapılan bir ön işleme aşamasını içerir.

- Ön işleme sırasında, grafikteki her düğümün önem sıralaması (importance rank) hesaplanır. Bu sıralama, düğümün grafiğin genel bağlantısındaki rolüne dayanır.
- Düğümün önemi, genellikle kenar farkı (edge difference) gibi metrikler kullanılarak belirlenir. Kenar farkı, bir düğüm kaldırıldığında eklenmesi gereken kenar sayısını ifade eder.
- Daha az önemli olan düğümler, kontrat edilir (yani kaldırılır), ancak bu düğümler üzerinden geçiş yapmak için gereken bağlantılar korunur.

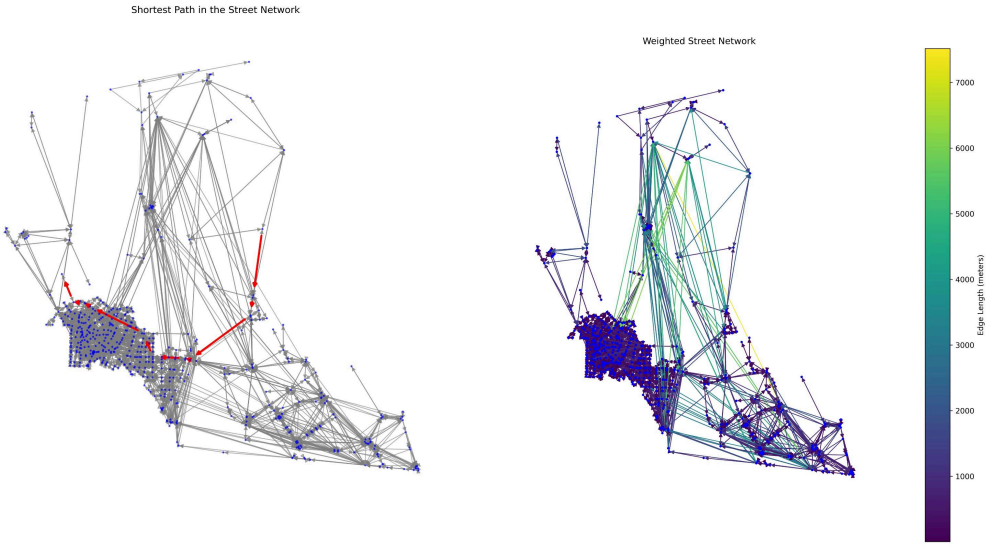
- Bu işlem sonucunda, düğümler arasındaki doğrudan bağlantılar korunur ve daha az önemli düğümlerin kaldırılmasıyla arama alanı küçültülür. Bu, sorgu sırasında algoritmanın çok daha hızlı çalışmasını sağlar.

```python
def preprocess(self):
    self.preprocessed = True
    node_pq = self.get_node_order_edge_difference()
    order = 0
    while node_pq:
        _, v = heapq.heappop(node_pq)
        new_dif = self.edge_difference(v)
        if node_pq and new_dif > node_pq[0][0]:
            heapq.heappush(node_pq, (new_dif, v))
            continue
        order += 1
        if (order % 500 == 0) or True:
            print(f".........Contracting {order}/{len(self.G.nodes)} nodes.........")
        self.order_of[v] = order
        self.node_order[order] = v

        for u in list(self.G.predecessors(v)):
            if u in self.order_of:
                continue
            P = {}
            for w in list(self.G.neighbors(v)):
                if w in self.order_of:
                    continue
                P[w] = self.times_all[u][v] + self.times_all[v][w]
            if not P:
                continue
            P_max = max(P.values())
            D = self.local_dijkstra_without_v(u, v, P_max)
            for w in list(self.G.neighbors(v)):
                if w in self.order_of:
                    continue
                if D[w] > P[w]:
                    if self.G.has_edge(u, w):
                        self.G.get_edge_data(u, w)[0]['shortcut_node'] = v
                    else:
                        self.G.add_edge(u, w, shortcut_node=v, length= P[w])
                    self.times_all[u][w] = P[w]
    print('Preprocess Done!')
```

get_shortest_path_CH(): Ön işleme tamamlandıktan sonra, bu yöntem en kısa yolu bulmak için çift yönlü Dijkstra algoritmasını kullanır, ancak contraction hierarchies ile arama alanını önemli ölçüde azaltır.

- Ön işleme aşamasında oluşturulan hiyerarşi, yol aramasında yalnızca önemli düğümlerin ve kenarların dikkate alınmasını sağlar.
- Sorgu sırasında, algoritma yalnızca en önemli düğümleri kontrol eder ve gereksiz düğümleri atlar.
- Bu yöntem, çok büyük grafiklerde bile en kısa yolu hızlı bir şekilde bulmak için oldukça etkilidir ve özellikle gerçek zamanlı navigasyon sistemlerinde yaygın olarak kullanılır.

```python
def get_shortest_path_CH(self, source_node, target_node):
    if not self.preprocessed:
        t1 = time.time()
        self.preprocess()
        t2 = time.time()
        print("Preprocessing time:", t2 - t1)
    t2 = time.time()
    t, p = self.bidirectional_dijkstra(source_node, target_node)
    t3 = time.time()
    if not p:
        return 0, p
    return t, p
```



Bu haritada Contraction Hierarchies (CH) algoritmasını kullanarak, okuldan Görükle'ye giden en kısa rotayı nasıl bulabileceğimizi göstereceğim. CH, özellikle büyük ağlarda sorgu sürelerini hızlandırmak için kullanılan bir algoritmadır. Algoritma, okul ve Görükle arasındaki yolu daha verimli bir şekilde bulmamıza olanak tanır.

Öncelikle, CH algoritması bir ön işleme aşaması yapar. Bu aşamada haritadaki düğümler, yani okul, Görükle ve aradaki kavşaklar gibi noktalar, önem sırasına göre sıralanır ve daha az önemli düğümler birleştirilerek 'atlama yolları' (shortcuts) oluşturulur. Örneğin, okuldan bir kavşağa ve ardından Görükle'ye olan ara yollar önceden hesaplanarak, sorgu sırasında gereksiz düğümlerin kontrol edilmesi engellenir.

Bu ön işlem tamamlandıktan sonra, sorgu aşaması başlar. Algoritma, okuldan Görükle'ye gitmek için hem ileri hem geri yönde bir Dijkstra araması yapar ve yalnızca önceden işlenmiş atlama yollarını kullanarak en kısa rotayı bulur. Bu yöntem, özellikle büyük harita ağlarında sorgulama süresini önemli ölçüde azaltır ve Görükle'ye en hızlı şekilde ulaşmamızı sağlar.

1.3 Simülasyon Sonucu Parametreler ile Karşılaştırma

Ön İşleme Süresi: Contraction Hierarchies (CH), hiyerarşik yapıyı oluşturmak için 1.74383 saniye gibi önemli bir ön işleme süresi gerektirir. Bu, daha hızlı sorgulamalar için bir defaya mahsus bir maliyettir.

Sorgu Süresi:

- Contraction Hierarchies: 0.0032 saniye (3.2 milisaniye)
- Dijkstra: 0.0156 saniye

Contraction Hierarchies, Dijkstra algoritmasından yaklaşık 5 kat daha hızlıdır.

İncelenen Düğüm Sayısı:

- Contraction Hierarchies: 58 üğüm
- Dijkstra: 439 düğüm

Contraction Hierarchies, Dijkstra'ya kıyasla yaklaşık 8 kat daha az düğüm incelemektedir.

1.4 Sonuç:

Contraction Hierarchies, tek seferlik bir ön işleme süresi yatırımıyla sorgu sırasında incelenen düğüm sayısını büyük ölçüde azaltır. Bu ön işleme aşamasında, haritadaki tüm düğümler analiz edilerek önem sırasına göre sıralanır ve daha az önemli olan düğümler birleştirilerek atlama yolları (shortcuts) oluşturulur. Bu sayede, sorgu sırasında algoritma yalnızca gerekli düğümleri inceleyerek en kısa rotayı çok daha hızlı bir şekilde hesaplar. Özellikle büyük yol ağlarında, her sorguda tüm düğümleri baştan incelemenin getirdiği zaman maliyetinden kaçınılır.

Bu yöntem, yoğun trafik ağları, büyük şehir haritaları veya sık sık en kısa yol hesaplaması gerektiren navigasyon uygulamaları gibi alanlarda Dijkstra algoritmasına kıyasla çok daha verimli bir seçenek olarak öne çıkar. Örneğin, bir şehirdeki yüzlerce kavşak ve yolun analiz edildiği bir ağda, Contraction Hierarchies algoritması, yalnızca önemli kavşakları ve bağlantıları dikkate alarak sorgu süresini büyük ölçüde hızlandırabilir. Bu hız avantajı, gerçek zamanlı navigasyon veya sık sorgu gerektiren sistemlerde önemli bir fark yaratır ve hem kullanıcı deneyimini hem de işlem verimliliğini artırır.

KAYNAKÇA

1- https://github.com/tungduong0708/Contraction-Hierarchy