一种有效实现用于解决无向稠密图中的所有成对最小最大化路径问题的方法

Gangli Liu
Tsinghua University
gl-liu13@mails.tsinghua.edu.cn

ABSTRACT

我们提供了一种高效的 $O(n^2)$ 实现，用于解决无向稠密图中的所有成对极小极大路径问题或最宽路径问题。这是之前论文中算法 4（通过计算和复制的 MMJ 距离）的一个代码实现。距离矩阵也称为所有点路径距离（APPD）。我们进行了实验以测试该实现和算法，并将其与其他几种用于解决 APPD 矩阵的算法进行了比较。结果显示，算法 4 对于求解极小极大路径或最宽路径问题的 APPD 矩阵工作良好。它可以大幅提高计算 APPD 矩阵的效率。有几个理论结果声称可以在 $O(n^2)$ 中准确地解决 APPD 矩阵。然而，它们并不实用，因为没有这些算法的代码实现。看起来算法 4 是第一个具有实际代码实现的算法，用于求解无向稠密图中极小极大路径或最宽路径问题的 APPD 矩阵，在 $O(n^2)$。

KEYWORDS

最小最大路径问题；最长腿路径距离；最小程序跳跃距离；最宽路径问题；最大容量路径问题；瓶颈边查询问题；所有点路径距离；弗洛伊德-沃舍尔算法；最小生成树

ACM Reference format:
Gangli Liu. 0000. 一种有效实现用于解决无向稠密图中的所有成对最小最大化路径问题的方法. In *Proceedings of 000, Beijing, China, 0000 (0000),* 6 pages.
DOI: 00.000/000_0

0000, Beijing, China
0000. 000-0000-00-000/00/00...$00.00
DOI: 00.000/000_0

1 介绍

极小极大路径问题是图论和优化中的一个经典问题。它涉及在加权图中找到一条路径，使得该路径上边的最大权重被最小化。[1]

给定一个图 $G = (V, E)$，其中 V 是顶点集，E 是边集，每条边 $e \in E$ 都有一个权重 e_w。对于一个有 n 个顶点的无向图，最大边数为 $\frac{n(n-1)}{2}$。一个稠密图接近有 $\frac{n(n-1)}{2}$ 条边。我们可以这么说，一个稠密图有 $O(n^2)$ 条边。在一个无向图中，每条边都是双向的，这意味着它在两个顶点之间以两种方向连接。最小化最大路径问题的目标是找到一条从起点 i 到终点 j 的路径 P，使得路径中边的最大权重 P 最小。两点之间的最小最大化路径距离是在这两点之间最小化最大路径中的最大权重（公式 2）。

$$\Phi = \{max\_weight(p) \mid p \in \Theta_{(i,j,G)}\} \tag{1}$$

$$M(i, j \mid G) = min(\Phi) \tag{2}$$

其中 G 是一个无向稠密图。$\Theta_{(i,j,G)}$ 是从节点 i 到节点 j 的所有路径的集合。p 是从节点 i 到节点 j 的路径，$max\_weight(p)$ 是路径 p 中的最大权重。Φ 是所有最大权重的集合。$min(\Phi)$ 是集合 Φ [16] 的最小值。

距离也可以被称为最长边路径距离（LLPD）[15] 或 Min-Max-Jump 距离（MMJ 距离）[16]。所有成对的极小极大路径距离计算数据集 X 中每一对点之间的距离或图 G 中的距离。它也被称为所有点路径距离

[1] https://en.wikipedia.org/wiki/Widest_path_problem

（APPD）[15]。它的形状是一个矩阵 $n \times n$。一个数据集 X 可以直接转换为一个完全图。

我们可以使用修改版的弗洛伊德-沃舍尔算法来解决有向图和无向稠密图 [20] 中的 APPD 问题，或者使用算法 1（递归 MMJ 距离）在 [16] 中，它们都需要 $O(n^3)$ 的时间。然而，在无向稠密图中，我们有更好的选择。我们可以使用一个 $O(n^2)$ 算法来计算 APPD 矩阵。有几个理论结果声称可以在 $O(n^2)$ [2, 8, 9, 19] 准确解决 APPD 矩阵问题。但是，这些算法没有代码实现，这意味着它们不切实际。代码实现是将设计或算法转化为编程语言的过程。在算法设计中这是至关重要的一步，因为想法被转化为可以执行特定任务的实际可执行代码。

在 [16] 的第 4.3 节（通过计算和复制的 MMJ 距离）中，作者提出了一种算法，该算法声称可以在 $O(n^2)$ 中准确解决 APPD 矩阵的问题，在无向稠密图中。该算法被称为算法 4（通过计算和复制的 MMJ 距离）。在论文中，该算法未被实现和测试。在这篇论文中，我们引入了算法 4 的代码实现，并对其进行了测试。

最宽路径问题是与最小最大路径问题密切相关的主题。相反，最宽路径问题的目标是找到从起始节点 s 到目标节点 t 的一条路径 P，使得该路径 P 上的边的最小权重最大化。最宽路径问题的任何算法都可以很容易地转换为解决最小最大路径问题的算法，反之亦然，只需通过反转算法执行的所有权重比较的意义即可。因此，我们可以粗略地说，最宽路径问题和最小最大路径问题是等价的。

2 相关工作

文献中提出了多种距离度量方法，包括欧几里得距离、曼哈顿距离、切比雪夫距离、闵可夫斯基距离、汉明距离和余弦相似性。这些度量在 k-NN、UMAP 和 HDBSCAN 等算法中经常被使用。欧氏距离是最常用的度量标准，而余弦相似性通常用于解决高维空间中的欧几里得距离问题。虽然欧几里得距离应用广泛且通用，但它不适应数据的几何结构，因为它与数据无关。因此，开发了各种依赖于数据的度量方法，例如源自数据集内扩散过程的扩散距离 [6, 7] 和基于路径的距离 [4, 10]。

最小最大路径距离已在各种机器学习模型中使用，如无监督聚类分析 [10–12, 15] 和有监督分类 [5, 16]。该距离通常在非凸和高度拉长的簇上表现良好，即使存在噪声 [15]。

2.1 最小最大化路径距离的计算

计算极小化最大路径距离的挑战在文献中被称为几个不同的名称，如最大容量路径问题、最宽路径问题、瓶颈边查询问题 [3, 13, 14, 17]、最长腿路径距离（LLPD）[15]，以及最小-最大化跳跃距离（MMJ 距离）[16]。直接计算极小化最大路径距离由于搜索空间庞大而计算成本高昂 [15]。然而，对于图 $G = G(V, E)$ 中固定的一对点 x 和 y 来说，距离可以在 $O(|E|)$ 时间内计算出来 [18]。关于最小最大路径距离的一个众所周知的事实是："最小生成树（MST）中任意两个节点之间的路径是最小最大路径。" [14] 利用这一结论，我们在计算最小最大路径距离时可以将无向稠密图简化为一棵最小生成树。

2.2 计算所有点路径距离

计算所有点的极小化极大路径距离被称为所有点路径距离（APPD）问题。将瓶颈生成树构造应用于每个点会导致 APPD 运行时间为 $O(\min\{n^2 \log(n)+n|E|, n|E| \log(n)\})$ [3, 13, 15] 。使用瓶颈生成树计算得出的 APPD 可能不准确，因为最小生成树（MST）必然也是最小瓶颈生成树（MBST），但最小瓶颈生成树不一定是最小生成树。Floyd-Warshall 算法的一个变体可以在 $O(n^3)$ [1] 时间内准确计算 APPD。几个理论结果表明，APPD 矩阵可以在 $O(n^2)$ 时间内准确求解 [2, 8, 9, 19]。然而，这些算法缺乏代码实现表明它们的不实用性。

3 算法的实现

As described in Section 1, the Algorithm 4 (MMJ distance by Calculation and Copy) in [16] also claims to solve the APPD matrix accurately in $O(n^2)$, in an undirected

Algorithm 4 MMJ distance by Calculation and Copy

Input: Ω
Output: $\mathbb{M}_\Omega$

1: **function** MMJ_Calculation_and_Copy(Ω)
2: Initialize $\mathbb{M}_\Omega$ with zeros
3: Construct a MST of Ω, noted T
4: Sort edges of T from large to small, generate a list, noted L
5: **for** e in L **do**
6: Remove e from T. It will result in two connected sub-trees, T_1 and T_2;
7: For all pair of nodes (p, q), where $p \in T_1$, $q \in T_2$. Fill in $\mathbb{M}_\Omega[p, q]$ and $\mathbb{M}_\Omega[q, p]$ with e.
8: **end for**
9: **return** $\mathbb{M}_\Omega$
10: **end function**

(a) 算法 4

```python
import networkx as nx

def cal_all_pairs_minimax_path_matrix_by_algo_4(distance_matrix):

    N = len(distance_matrix)
    all_pairs_minimax_matrix = np.zeros((N,N))

    MST = construct_MST_from_graph(distance_matrix)

    MST_edge_list = list(MST.edges(data='weight'))


    edge_node_list = [(edge[0],edge[1]) for edge in MST_edge_list]
    edge_weight_list = [edge[2] for edge in MST_edge_list]

    edge_large_to_small_arg = np.argsort(edge_weight_list)[::-1]

    edge_weight_large_to_small = np.sort(edge_weight_list)[::-1]
    edge_nodes_large_to_small = [edge_node_list[i] for i in edge_large_to_small_arg]

    for i, edge_nodes in enumerate(edge_nodes_large_to_small):
        edge_weight = edge_weight_large_to_small[i]
        MST.remove_edge(*edge_nodes)

        tree1_nodes = list(nx.dfs_preorder_nodes(MST, source=edge_nodes[0]))
        tree2_nodes = list(nx.dfs_preorder_nodes(MST, source=edge_nodes[1]))

        for p1 in tree1_nodes:
            for p2 in tree2_nodes:
                all_pairs_minimax_matrix[p1, p2] = edge_weight
                all_pairs_minimax_matrix[p2, p1] = edge_weight

    return all_pairs_minimax_matrix
```

(b) Python 实现的算法 4

图 1: 算法 4 及其 Python 实现。三个嵌套的 for 循环使它看起来像一个 $O(n^3)$ 算法，但实际上它是一个 $O(n^2)$ 算法。

Implementation ID	Implementation name	Complexity	Coding language	Notes
0	Algo_1_Python	$O(n^3)$	Python	Algorithm 1 (MMJ distance by recursion)
1	Algo_1_C++	$O(n^3)$	C++	Algorithm 1 (MMJ distance by recursion)
2	Floyd_Warshall_Python	$O(n^3)$	Python	A variant of Floyd-Warshall Algorithm
3	Floyd_Warshall_C++	$O(n^3)$	C++	A variant of Floyd-Warshall Algorithm
4	MST_shortest_path	$O(n^3 log(n))$	Python	Calculate the shortest path in a MST
5	Algo_4	$O(n^2)$	Python	Algorithm 4 (MMJ distance by Calculation and Copy)

表 1: 四个算法的概况。其中两个用不同的编程语言实现，分别是 Python 和 C++。

	data 139 (N = 120)	data 109 (N = 300)	data 18 (N = 500)	data 19 (N = 850)	data 16 (N = 2500)	data 35 (N = 5000)	data 136 (N = 10000)
Algo_1_Python	13.451s	208.363s	990.308s	4681.911s	>7200s	>7200s	>7200s
Algo_1_C++	0.033s	0.414s	1.794s	9.032s	237.961s	1986.928s	>7200s
Floyd_Warshall_Python	1.489s	23.353s	106.745s	534.683s	>7200s	>7200s	>7200s
Floyd_Warshall_C++	0.033s	0.436s	2.324s	10.035s	253.909s	2162.514s	>7200s
MST_shortest_path	0.399s	4.229s	24.926s	110.449s	2503.483s	>7200s	>7200s
Algo_4	0.02s	0.073s	0.191s	0.511s	4.311s	17.015s	67.048s

表 2: 四个算法的性能。N 是数据集中的点数。

dense graph. But it is left unimplemented and untested. Figure 1a is Algorithm 4 (MMJ distance by Calculation and Copy) in [16], for convenience of reading, we re-post it here. Figure 1b is its python implementation.

注意三个嵌套的 for 循环使其看起来像一个 $O(n^3)$ 算法，但实际上它是一个 $O(n^2)$ 算法。因为当变量 i 在 *Line* 21 中较小时，树 *1* 和树 *2* 的大小均为 $O(n)$；但当变量 i 较大时，树 *1* 和树 *2* 的大小均为 $O(1)$。最终的净效果是三个嵌套的 for 循环只访问 APPD 矩阵中的每个单元格一次。因此，它是一个 $O(n^2)$ 算法。

在实现过程中，我们首先构造一个无向稠密图的最小生成树（MST）。使用 prim 算法构建 MST 的复杂度是 $O(n^2)$。然后，我们将 MST 中的边按降序排序。从大到小依次删除 MST 中的边是非常关键的。只有

```python
# G is an undirected dense graph, which has N vertices.
# adj_matrix is its adjacency_matrix.

def variant_of_Floyd_Warshall(adj_matrix):
    p = adj_matrix.copy()
    N = len(adj_matrix)

    for i in range(N):
        for j in range(N):
            if i != j:
                for k in range(N):
                    if i != k and j != k:
                        p[j,k] = min (p[j,k], max (p[j,i], p[i,k]))
    return p
```

图 2: Floyd-Warshall 算法的一种变体用于解决最小化最大路径问题

```python
# G is an undirected dense graph, which has N vertices.
import networkx as nx
def MST_shortest_path(G):

    MST = nx.minimum_spanning_tree(G)
    minimax_matrix = np.zeros((N, N))

    for i in range(N):
        for j in range(N):
            if j > i:
                max_weight = -1
                path = nx.shortest_path(MST, source=i, target=j)
                for k in range(len(path)-1):
                    if( MST.edges[path[k],path[k+1]]['weight'] > max_weight):
                        max_weight = MST.edges[path[k],path[k+1]]['weight']
                minimax_matrix[i,j] = minimax_matrix[j,i] = max_weight

    return minimax_matrix
```

图 3: Python 实现的最小生成树_最短路径，参见表 1

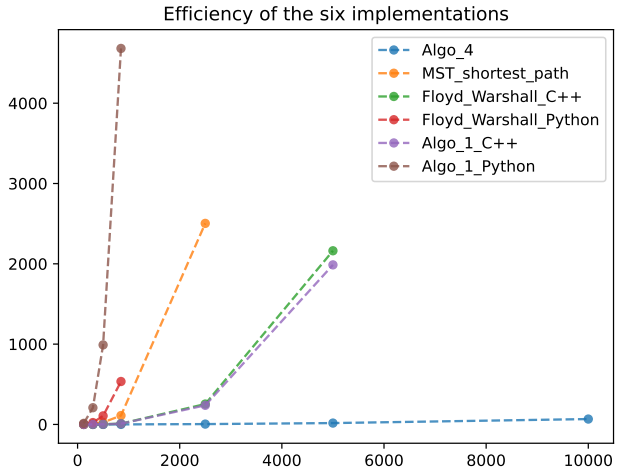

图 4: 算法（实现）的性能

这样我们才能得到两个子树，树 *1* 和树 *2*。通过遍历每个子树，可以分别获得这两个子树中的节点。

进一步的探索表明，我们也可以通过首先将边按升序排列（从小到大），然后依次将边添加到一个空的 MST 树中来计算 APPD 矩阵。[2]

4 算法的测试

在一个实验中，我们在七个具有不同数据点数量的数据集上测试了算法 4（通过计算和复制获得 MMJ 距离）。请注意，一个数据集可以很容易地转换为一个完整图。算法 4 的性能与其他三种能够计算 APPD 矩阵的算法进行了比较。

表 1 列出了四个算法的概况。*算法_1* 是 [16] 中的算法 1（递归计算 MMJ 距离），其复杂度为 $O(n^3)$；*弗洛伊德-沃舍尔算法* 是 Floyd-Warshall 算法的一种变体。图 2 是它的 Python 实现。它的时间复杂度为 $O(n^3)$；*最小生成树_最短路径* 首先构建无向稠密图的最小生成树（MST），然后计算每对节点之间的最短路径，接着计算最短路径上的最大权重。其时间复杂度为 $O(n^3 log(n))$。图 3 是它的 Python 实现。实现基于 Madhav-99 的代码 [3]；*算法_4* 是 [16] 中的算法 4（通过计算和复制计算 MMJ 距离），其复杂度为 $O(n^2)$。*算法_1* 和 *弗洛伊德-沃舍尔* 分别使用 C++ 和 python 实现，以测试不同编程语言之间的差异。

4.1 性能

表 2 是算法（实现）的性能。我们用七个具有不同数据点数量的数据集测试每个算法。与数据 ID 对应的数据源可以在以下 URL 找到。[4] 数值是在配备 “3.3 GHz Quad-Core Intel Core i5” CPU 和 16 GB RAM 的台式计算机上，通过每种算法计算 minimax 路径 APPD 所需的时间。

为了节省时间，如果算法在 7200 秒（两小时）内无法获得 APPD 矩阵，则停止该算法的执行。每个数据集和算法的计算时间仅记录一次。图 4 将表 2 中的值转换为图形。可以看出，算法 4 比其他算法表现更

[2] 参见 “按升序排列 MST 的边.ipynb” 文件在 https://github.com/mike-liuliu/Algorithm_4 中

[3] https://github.com/Madhav-99/Minimax-Distance

[4] https://github.com/mike-liuliu/Min-Max-Jump-distance

好。它可以大约在 67 秒内计算出 10,000 个点的 APPD 矩阵，而其他算法则无法在两小时内完成。

合理地，*算法_1* 和*弗洛伊德-沃舍尔算法*的 C++ 实现比它们的 Python 版本快得多。有趣的是，在使用 Python 实现时，*算法_1* 比*弗洛伊德-沃舍尔算法*慢得多，但在 C++中比*弗洛伊德-沃舍尔算法*稍快一些。

4.2 求解最宽路径问题

正如第 7 节（解决最宽路径问题）在 [16] 中所述，算法 4（通过计算和复制求 MMJ 距离）可以通过构建最大生成树并将边按升序排列来修改以解决无向图中的最宽路径问题。在另一个实验中，我们测试了使用算法 4 计算最宽路径 APPD。结果显示算法 4 很好地解决了最宽路径问题。

5 算法的证明

一个好的问题是算法 4（计算和复制的 MMJ 距离）为什么有效。这里是对该算法正确性的理论证明。

每当我们要从最小生成树中移除一条边 e 时，e 必须属于 MST 的一个连通子树 T。该子树记为 S_t。子树是完全包含在另一个树中的树。注意最小生成树 T 可以被视为其自身的子树。我们可以得出边 e 是子树 S_t 中最大的边。由于边已经按降序排列，并且在之前的步骤中移除了大于 e 的边。如果在 S_t 中存在其他与 e 一样大的边，这并不重要。从 S_t 中移除边 e 后，我们得到两个较小的连通子树，$tree1$ 和 $tree2$。对于任意一对节点 (p, q)，其中 $p \in tree1$,$q \in tree2$，节点 p 和 q 之间的极小极大路径距离必须是边 e 的权重。因为“最小生成树（MST）中任意两个节点之间的路径是最小最大路径” [14]，且边 e 是子树 S_t 中的最大边。从 p 到 q 的路径必须经过边 e，且边 e 是在该路径上的最大边。路径中是否有其他边与 e 一样大并不重要。注意，仅有一个节点的子树被视为有效的子树。

因此，p 和 q 之间的极小极大路径距离必须是边 e 的权重。算法 4（通过计算和复制的 MMJ 距离）的正确性得到了证明。

6 讨论

6.1 算法 1 的优点

算法 1（递归的 MMJ 距离）具有热启动的优点。假设我们已经计算出了大型图 G 的 APPD 矩阵 M_G，然后得到了一个新的点（或节点）p，其中 $p \notin G$。新的图记为 $G + p$。要计算图 $G + p$ 的 APPD 矩阵，如果我们使用其他算法，可能需要从零开始。算法 1 的优点在于利用计算出的 M_G 来计算新的 APPD 矩阵，并且结合了定理 3.3、3.5、6.1 和推论 3.4 中的结论，这些内容在 [16] 中有所阐述。这尤其适用于有向稠密图的情况，在这种情况下从零开始需要 $O(n^3)$ 的复杂度，而算法 1（通过递归计算 MMJ 距离）的热启动仅需 $O(n^2)$ 的复杂度。我们可以说算法 1 支持在线机器学习 [5]，在其中数据以顺序形式逐步提供。

6.2 使用并行编程

如果计算 APPD 矩阵的主要关注点是速度，我们可以使用并行编程来加速算法 4。首先，我们可以使用不同的处理器遍历图 1b 中 *Line* 25 *and* 26 的树 *1* 和树 *2*。其次，我们可以将最小生成树（MST）复制到许多处理器上。对于第 n 个处理器，我们只需移除最大的 n 条边，得到第 n 个树 *1* 和树 *2*，遍历它们，然后填充由第 n 个树 *1* 和树 *2* 决定的 APPD 矩阵中的相应位置。

算法 4 的并行加速版本（通过计算和复制计算 MMJ 距离），称为算法 13（由并行计算加速的 APPD）。这里 [6] 是算法 13 四种变体的 C++代码实现。

7 结论

我们实现了之前论文中介绍的算法 4（通过计算和复制的 MMJ 距离）。然后测试了该实现，并将其与几种其他可以计算所有成对最小最大路径距离或也称为所有点路径距离（APPD）的算法进行了比较。实验表明，算法 4 在解决最宽路径或最小最大路径 APPD 矩阵方面效果良好。作为一个复杂度为 $O(n^2)$ 的算法，它可

[5] https://en.wikipedia.org/wiki/Online_machine_learning
[6] https://github.com/mike-liuliu/Algorithm_4/tree/main/Algorithm_4_in_cpp_parallel_computing

以极大地提高计算 APPD 矩阵的效率。注意，用于求解 APPD 矩阵的算法至少具有 $O(n^2)$ 的复杂度，因为该矩阵是一个 $n \times n$ 矩阵。

在论文《基于路径的光谱聚类：保证、对异常值的鲁棒性和快速算法》的第 2.3.3 节中，[15]Murphy 博士和他的合作者写道：

简单地将瓶颈生成树构造应用于每个点会得到一个运行时间为 $O(min\{n^2log(n) + n|E|, n|E|log(n)\})$ 的 *APPD*。然而，可以通过修改后的 *SLINK* 算法（*Sibson*，*1973*）或笛卡尔树（*Alon* 和 *Schieber*，*1987*；*Demaine* 等人，*2009*，*2014*）在 $O(n^2)$ 时间内计算出 *APPD* 距离矩阵。

作者发送了一封电子邮件以进一步澄清此陈述。

作者：

你指出可以计算 $O(n^2)$ 中的 *APPD* 距离矩阵。然而，我在互联网和 *github* 上搜索后，并未找到任何能够准确计算 $O(n^2)$ 中 *APPD* 距离矩阵的代码实现。你知道有这样的代码实现吗？请告诉我。

墨菲博士：

如果你能找到一个在 $O(n^2)$ 中实现 *SLINK* 来进行单链聚类的方法，那么你可以通过从生成的系统树中读取距离来完成 *APPD*。我不知道有任何 *SLINK* 的实现，并且可能对它进行理论证明比实际实现更容易。

关于树结构，这些无疑更多是理论兴趣所在，我不奇怪它们可能根本没有实际应用。因此，通过那些方法实现 $O(n^2)$ 可能不切实际。

因此，我们可以大致得出结论：在 [16] 中的算法 4（通过计算和复制得到的 MMJ 距离）是第一个具有实际代码实现来解决未定向稠密图中最小最大路径或最宽路径问题的算法，在 $O(n^2)$ 中。

REFERENCES

[1] Alfred V Aho and John E Hopcroft. 1974. *The design and analysis of computer algorithms*. Pearson Education India.

[2] Noga Alon and Baruch Schieber. 2024. Optimal preprocessing for answering on-line product queries. *arXiv preprint arXiv:2406.06321* (2024).

[3] Paolo M. Camerini. 1978. The min-max spanning tree problem and some extensions. *Inform. Process. Lett.* 7, 1 (1978), 10–14.

[4] Hong Chang and Dit-Yan Yeung. 2008. Robust path-based spectral clustering. *Pattern Recognition* 41, 1 (2008), 191–203.

[5] Morteza Haghir Chehreghani. 2017. Classification with minimax distance measures. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 31.

[6] Ronald R Coifman and Stéphane Lafon. 2006. Diffusion maps. *Applied and computational harmonic analysis* 21, 1 (2006), 5–30.

[7] Ronald R Coifman, Stephane Lafon, Ann B Lee, Mauro Maggioni, Boaz Nadler, Frederick Warner, and Steven W Zucker. 2005. Geometric diffusions as a tool for harmonic analysis and structure definition of data: Diffusion maps. *Proceedings of the national academy of sciences* 102, 21 (2005), 7426–7431.

[8] Erik D Demaine, Gad M Landau, and Oren Weimann. 2009. On cartesian trees and range minimum queries. In *Automata, Languages and Programming: 36th International Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part I 36*. Springer, 341–353.

[9] Erik D Demaine, Gad M Landau, and Oren Weimann. 2014. On cartesian trees and range minimum queries. *Algorithmica* 68 (2014), 610–625.

[10] Bernd Fischer and Joachim M. Buhmann. 2003. Path-based clustering for grouping of smooth curves and texture segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 25, 4 (2003), 513–518.

[11] Bernd Fischer, Volker Roth, and Joachim Buhmann. 2003. Clustering with the connectivity kernel. *Advances in neural information processing systems* 16 (2003).

[12] Bernd Fischer, Thomas Zöller, and Joachim M Buhmann. 2001. Path based pairwise data clustering with application to texture segmentation. In *Energy Minimization Methods in Computer Vision and Pattern Recognition: Third International Workshop, EMMCVPR 2001 Sophia Antipolis, France, September 3–5, 2001 Proceedings 3*. Springer, 235–250.

[13] Harold N Gabow and Robert E Tarjan. 1988. Algorithms for two bottleneck optimization problems. *Journal of Algorithms* 9, 3 (1988), 411–417.

[14] TC Hu. 1961. The maximum capacity route problem. *Operations Research* 9, 6 (1961), 898–900.

[15] Anna V. Little, Mauro Maggioni, and James M. Murphy. 2020. Path-Based Spectral Clustering: Guarantees, Robustness to Outliers, and Fast Algorithms. *J. Mach. Learn. Res.* 21 (2020), 6:1–6:66. http://jmlr.org/papers/v21/18-085.html

[16] Gangli Liu. 2023. Min-Max-Jump distance and its applications. *arXiv preprint arXiv:2301.05994* (2023).

[17] Maurice Pollack. 1960. The maximum capacity through a network. *Operations Research* 8, 5 (1960), 733–736.

[18] Abraham P Punnen. 1991. A linear time algorithm for the maximum capacity path problem. *European Journal of Operational Research* 53, 3 (1991), 402–404.

[19] Robin Sibson. 1973. SLINK: an optimally efficient algorithm for the single-link cluster method. *The computer journal* 16, 1 (1973), 30–34.

[20] Eric W Weisstein. 2008. Floyd-warshall algorithm. *https://mathworld.wolfram.com/* (2008).