1 Probability of Path Collisions

We consider a random communication pattern of endpoints, where for each endpoint s', we pick an endpoint t' uniformly at random. This also gives us, u.a.r., a vertex t_i for each s'_i attached to some vertex s, effectively we get $N = p \cdot N_r$ pairs (s,t) sampled u.a.r.. Now, the probability of having a path collision is given by the birthday paradox, with the number of node pairs $P = N_r^2$ as the number of days and the number of endpoints N as the sample size[1]:

$$\mathscr{P}(\text{collision}) = 1 - \frac{P^{\underline{N}}}{P^N} \tag{1}$$

For the given random pattern, this is essentially 1 for all usual configurations, even with $p = 1$ it is $\approx 50\%$. We can also look at smaller subsets of endpoints, e. g. the same pattern for all sources on a single node (note that now P is smaller): for $p = \frac{k'}{2}$, we get around 5% ($N_r = 722$, SF). This means collisions are not negligible even for low-load, random patterns.

We can also compute the expected number of colliding paths C (summed over all colliding sets, excluding the first, non-colliding, path per set) for this workload:

$$E(C) = N - P + P\left(\frac{P-1}{P}\right)^N \tag{2}$$

This gives values of a few times k' for the SF configurations with $p = \frac{k'}{2}$.

For comparison, the number of collisions on endpoints ($s'_1 \neq s'_2$ with $t'_1 = t'_2$) is according to the same formula

$$E(C') = N - N + N\left(\frac{N-1}{N}\right)^N \approx \frac{N}{e} \tag{3}$$

which is much higher and impacts performance much more.

We can also estimate the maximum size of a collision group: the number of collision groups is bounded by the total number of colliding paths. Since this is small compared to P, we can estimate the probability of a size three set as

$$\mathscr{P}(\text{3-way collision}) \approx 1 - \left(1 - \frac{C}{P}\right)^C \tag{4}$$

which remains around 1% for the mentioned configuration.

Therefore, we can conclude that multi-path routing has to be able to handle at least pairs of colliding paths on a small subset of paths. If there are no shortest paths available, non-minimal paths can be used and the average path length will increase slightly.

2 Efficient Path Counting

Some of the measures for path diversity are computationally hard to derive for large graphs. Algorithms for all-pairs shortest paths analysis based on adjacency matrices are well known, and we reintroduce one such method here for the purpose of reproducibility. For the disjoint-paths analysis however, all-pairs algorithms exist, but are not commonly known. We introduce a method by Cheung et. al [**?**] and *we adapt for length-limited edge connectivity computation*.

2.1 Matrix Multiplication for Path Counting

It is well known that for a graph represented as an adjacency matrix, matrix multiplication (MM) can be used to obtain information about paths in that graph. Variations of this include the Floyd-Warshall algorithm [**?**] for transitive closure and all-pairs shortest paths [**?**], which use different semirings to aggregate the respective quantities. To recapitulate how these algorithms work, consider standard MM using $\cdot$ and $+$ operators on non-negative integers, which computes the number of paths $n_i(s,t)$ between each pair of vertices.

Theorem 1. *If A is the adjacency matrix of a directed graph $G = (V,E)$, $A_{i,j} = 1$ iff $(i,j) \in E$ and $A_{i,j} = 0$ iff $(i,j) \notin E$, then each cell $i \in V, j \in V$ of $Q = A^l = \underbrace{A \cdot \ldots \cdot A}_{l \text{ times}}$ contains the number of paths from i to j with exactly l steps in G.*

Proof. By induction on the path length l: For $l = 1$, $A^l = A$ and the adjacency matrix contains a 1 in cell i,j iff $(i,j) \in E$, else 0. Since length-1 paths consist of exactly one edge, this satisfies the theorem. Now consider matrices A^p, A^q for $p + q = l$ for which the theorem holds since $p,q < l$. We now prove the theorem also holds for $A^l = A^p \cdot A^q$. Matrix multiplication is defined as

$$(A^p \cdot A^q)_{i,j} = \sum_k A^p_{i,k} \cdot A^q_{k,j}. \tag{5}$$

According to the theorem, $A^p_{i,k}$ is the number of length-p paths from i some vertex k, and $A^q_{k,j}$ is the number of length-q paths from said vertex k to j. To reach j from i via k, we can choose any path from i to k and any form k to j, giving $A^p_{i,k} \cdot A^q_{k,j}$ options. Since we are interested in *all* paths from i to j, we consider *all* intermediate vertices k and count the total number (sum) of paths. This is exactly the number of length-l paths demanded by the theorem, since each length-l path can be uniquely split into a length-p and a length-q segment. □

In the proof we ignored a few details caused by the adjacency matrix representation: first, the adjacency matrix models a directed graph. We can also use the representation for undirected graphs by making sure A is symmetrical (then also A^l is symmetrical). Adjacency matrices contain the entry $A_{i,j} = 0$ to indicate $(i,j) \notin E$ and $A_{i,j} = 1$ for $(i,j) \in E$. By generalizing $A_{i,j}$ to be the number of length-1 paths (= number of edges) from i to j as in the theorem, we can also represent multi-edges; the proof still holds.

Finally, the diagonal entries $A_{i,i}$ represent self-loops in the graph, which need to be explicitly modeled. Note that also

[1] $x^{\underline{k}}$ is the falling factorial $x \cdot (x-1) \cdot \ldots \cdot (x-k+1)$

$i = j$ is allowed above and the intermediate vertex k can be equal to i and/or j. Usually self-loops should be avoided by setting $A_{i,i} = 0$. Then $A^l_{i,i}$ will be the number of cycles of length l passing through i, and the paths counted in $A_{i,j}$ will include paths containing cycles. These cannot easily be avoided in this scheme[2]. For most measures, e.g., shortest paths or disjoint paths, this is not a problem, since paths containing cycles will naturally never affect these metrics.

On general graphs, the algorithms outlined here are not attractive since it might take up to the maximum shortest path length D iterations to reach a fixed point, however since we are interested in low-diameter graphs, they are practical and easier to reason about than the Floyd-Warshall algorithms.

2.1.1 Matrix Multiplication for Routing Tables

As another example, we will later use a variation of this algorithm to compute next-hop tables that encode for each source s and each destination t which out-edge of s should be used to reach t. In this algorithm, the matrix entries are sets of possible next hops. The initial adjacency matrix will contain for each edge in G a set with the out edge index of this edge, otherwise empty sets. Instead of summing up path counts, we union the next-hop sets, and instead of multiplying with zero or one for each additional step, depending if there is an edge, we retain the set only if there is an edge for the next step. Since this procedure is not associative, it cannot be used to form longer paths from shorter segments, but it works as long as we always use the original adjacency matrix on the right side of the multiplication. The correctness proof is analogous to the path counting procedure.

2.2 Counting Disjoint Paths

The problem of counting all-pairs disjoint paths per pair is equivalent to the all-pairs edge connectivity problem which is a special case of the all-pairs max flow problem for uniform edge capacities. It can be solved using a spanning tree (*Gomory-Hu tree* [**?**]) with minimum $s-t$-cut values for the respective partitions on the edges. The minimum $s-t$ cut for each pair is then the minimum edge weight on the path in this tree, which can be computed cheaply for all pairs. The construction of the tree requires $\mathcal{O}(N_r)$ $s-t$-cuts, which cost $\mathcal{O}(N_r^3)$ each (e.g., using the Push-Relabel scheme [**?**]).

Since we are more interested in the max flow values, rather than the min-cut partitions, a simplified approach can be used: while the Gomory-Hu tree has max flow values and min cut partitions equivalent to the original graph, a *equivalent flow tree* [**?**] only preserves the max flow values. While constructing it needs the same number of max-flow computations, these can be performed on the original input graph rather than the contracted graphs of Gomory-Hu, which makes the implementation much easier.

For length-restricted connectivity, common max-flow algorithms have to be adapted to respect the path length constraint. The Gomory-Hu approach does not work, since it is based on the principle that the distances in the original graph do not need to be respected. We implemented an algorithm based on the Ford-Fulkerson method [**?**], using Breadth-First Search (BFS) [**?**], which is not suitable for an all-pairs analysis, but can provide results for small sets of samples.

The spanning-tree based approaches only work for undirected graphs, and solve the more general max-flow problem. There are also algorithms that only solve the edge-connectivity problem, using completely different approaches. Cheung et. al [**?**] propose an algorithm based on linear algebra which can compute all-pairs connectivity in $\mathcal{O}(|E|^\omega + |V|^2 k'^\omega)$ where $\omega \leq 3$ is the exponent for matrix-matrix multiplication. For our case of $k' \approx \sqrt{N_r}$ and naive matrix inversion, this is $\mathcal{O}(N_r^{4.5})$ with massive space use, but there are many options to use sparse representations and iterative solvers, which might allow to reach $\mathcal{O}(N_r^{3.5})$. Due to their construction, those algorithms also allow a limitation of maximum path length (with a corresponding complexity reduction) and the heavy computations are built on well-known primitives with low constant overhead and good parallel scaling, compared to classical graph manipulation.

2.3 Deriving Edge Connectivity

This scheme is based on the ideas of Cheung et. al. [**?**]. First we adapt the algorithm for vertex connectivity, which allows lower space- and time complexity than the original algorithm and might also be easier to understand. The original edge-connectivity algorithm is obtained by applying it to a transformed graph.[3] We then introduce the path-length constraint by replacing the exact solution obtained by matrix inversion with an approximated one based on iterations, which correspond to incrementally adding steps. The algorithm is randomized in the same way as the original is; we will ignore the probability analysis for now, as the randomization is only required to avoid degenerate matrices in the process and allow the use of a finite domain. The domain $\mathbb{F}$ is defined to be a finite field of sufficient size to make the analysis work and allow a real-world implementation; we can assume $\mathbb{F} = \mathbb{R}^+$ for the algorithm itself.

First, we consider a *connection matrix*, which is just the adjacency matrix with random coefficients for the edges:

$$K_{i,j} = \begin{cases} x \in \mathbb{F} \text{ u.a.r.} & \text{iff } (i,j) \in E \\ 0 & \text{else}. \end{cases} \tag{6}$$

In the edge-connectivity algorithm we use a much larger adjacency matrix of a transformed graph here (empty rows and columns could be dropped, leaving an $|E| \times |E|$ matrix,

[2] Setting $A^l_{i,i} = 0$ before/after each step does not prevent cycles, since a path from i to k might pass j, causing a cycle, and we cannot tell this is the case without actually recording the path.

[3] Vertex-connectivity, defined as the minimum size of a cut set $c_{st} \subset V \setminus \{s,t\}$ of vertices that have to be removed to make s and t disconnected, is not well defined for neighbors in the graph. The edge-connectivity algorithm avoids this problem, but this cannot be generalized for vertex-connectivity.

but our implementation does not do this since the empty rows and columns are free in a sparse matrix representation):

$$K'_{(i,k),(k,j)} = \begin{cases} x \in \mathbb{F} \text{ u.a.r.} & \text{iff } (i,k) \in E \wedge (k,j) \in E \\ 0 & \text{else}. \end{cases} \tag{7}$$

Now, we assign a vector $F_i \in \mathbb{F}^k$, where k is the maximum vertex degree, to each vertex i and consider the system of equations defined by the graph: the value of each vertex shall be the linear combination of its neighbors weighted by the edge coefficients in K. To force a non-trivial solution, we designate a source vertex s and add pairwise orthogonal vectors to each of its neighbors. For simplicity we use unit vectors in the respective columns of a $k \times |V|$ matrix P_s (same shape as F). So, we get the condition

$$F = FK + P_s\,. \tag{8}$$

This can be solved as

$$F = -P_s(\mathbb{I} - K)^{-1}\,. \tag{9}$$

The work-intensive part here is inverting $(\mathbb{I} - K)$, which can be done explicitly and independently from s, to get a computationally inexpensive all-pairs solution, or implicitly only for the vectors in P_s for a computationally inexpensive single-source solution. To compute connectivity, we use the following theorem. The algorithm outlines in the following proof counts vertex-disjoint paths of any length.

Theorem 2. *The size of the vertex cut set c_{st} from s to t equals* $\operatorname{rank}(FQ_t)$*, where $F = -P_s(\mathbb{I}-K)^{-1}$ and Q_t is a $|V| \times k$ permutation matrix selecting the incoming neighbors of t.*

Proof. First, $c_{st} \leq \operatorname{rank}(FQ_t)$, because all non-zero vectors were injected around s and all vectors propagated through the cut set of c_{st} vertices to t, so there cannot be more than c_{st} linearly independent vectors near t. Second, $c_{st} \geq \operatorname{rank}(FQ_t)$, because there are c_{st} vertex-disjoint paths from s to t. Each passes through one of the c_{st} outgoing neighbors of s, which has one of the linearly independent vectors of P_s assigned (combined with potentially other components). As there is a path from s to t trough this vertex, on each edge of this path the component of P_s will be propagated to the next vertex, multiplied by the respective coefficient in K. So, at t each of the paths will contribute one orthogonal component. □

To count length-limited paths instead, we simply use an iterative approximation of the fixed point instead of the explicit solution. Since we are only interested in the ranks of sub-matrices, it is also not necessary to actually find a precise solution; rather, following the argument of the proof above, we want to follow the propagation of linearly independent components through the network. The first approach is simply iterating Equation **??** from some initial guess. For this guess we use zero vectors, due to P_s in there we still get nontrivial solutions but we can be certain to not introduce additional linearly dependent vectors:

$$\begin{aligned} F_0 &= \begin{pmatrix} 0 \end{pmatrix} \quad (k \times |V|) \\ F_l &= F_{l-1}K + P_s\,. \end{aligned} \tag{10}$$

This iteration still depends on a specific source vertex s. For an all-pairs solution, we can iterate for all source vertices in parallel by using more dimensions in the vectors, we set $k = |V|$. Now we can assign every vertex a pairwise orthogonal start vector, e.g., by factoring out P_s and selecting rows by multiplying with P_s in the end. The intermediate products are now $|V| \times |V|$ matrices, and we add the identity matrix after each step. Putting the parts together, we obtain

$$c_{st} = \operatorname{rank}(P_s \underbrace{(((K+\mathbb{I})\cdot K + \mathbb{I})\cdot \ldots)}_{l \text{ times, precomputed}} Q_t)\,. \tag{11}$$

The total complexity includes the $\mathcal{O}\left(|V|^3 l\right)$ operations to precompute the matrix for a maximum path length of l and $\mathcal{O}\left(|V|^2 k^3\right)$ operations for the rank operations for all vertex pairs in the end, which will be the leading term for the $k = \mathcal{O}\left((\sqrt{|V|}\right)$ (diameter 2) undirected graphs considered here, for a total of $\mathcal{O}\left(|V|^{3.5}\right)$.

For the edge connectivity version, we use the edge incidence connection matrix K', and select rows and columns based on edge incidence, instead of vertex adjacency. Apart from that, the algorithm stays identical, but the measured cut set will now be a cut set of edges, yielding edge connectivity values. However, the algorithm is more expensive in terms of space use and running time: $\mathcal{O}\left(|E|^3 l\right)$ to precompute the propagation matrix.