Hard unknots are often easy from a different perspective

Jason Cantarella[1], Henrik Schumacher[2], and Clayton Shonkwiler[3]

[1]Mathematics Department, University of Georgia, Athens, GA, USA
[2]Institute for Mathematics, RWTH Aachen University, Aachen, Germany
[3]Department of Mathematics, Colorado State University, Fort Collins, CO, USA

August 4, 2026

Abstract

Recent attempts to train AI models to recognize knots have produced millions of "hard" unknot diagrams resistant to simplification by Reidemeister moves, pass moves, or random walks on the Reidemeister graph. Most are easy for non-diagrammatic methods such as simplifying triangulations of the knot complement (`Regina`) or presentations of the knot group (`SnapPy`). We present `ReAPR` (Re-embedding And Pass Rerouting), which alternates pass-move reduction with a geometric re-embedding step. The re-embedding minimizes the *total variation* of a height function on the diagram subject to crossing constraints. We show that for an n-crossing diagram, the minimum total variation is $2(n-k)$, where k is the least number of crossings one must virtualize to make the diagram virtually alternating; this is a combinatorial invariant of the diagram. Reprojecting the resulting embedding from a new viewpoint reveals previously hidden simplifications. `ReAPR` successfully simplifies every published hard-unknot example we are aware of, as well as several new collections ($\approx$ 2.6 million examples in total) in under 30 seconds of total CPU time. This includes a set of Kauffman's "challenge" unknots, presented as rational tangles, which appear to be surprisingly difficult for both non-diagrammatic methods.

1. Introduction

Given a knot diagram, an interesting computational problem is to transform it to a diagram of the same knot with as few crossings as possible. Deciding whether such a simplification exists, and finding one if so, are problems that go back to Turing [1]. This problem has recently attracted a lot of new interest as attempts are made to develop AI methods to recognize and classify knots [2–13]. It is desirable to have a method for this simplification step which is extremely effective, extremely fast, and scalable to very large diagrams. We present `ReAPR` (Re-embedding And Pass Rerouting), which is the first simplifier to reach all three goals.

To establish some notation, we think of an n-crossing knot diagram D as a directed, 4-valent, embedded planar graph $G(D)$ with over and under information at the crossings. The edges of this graph are *arcs* of the knot $a_0, \ldots, a_{2n-1}$; we assume that they are numbered in the order in which we visit them as we follow the knot. A (maximal) *overpass* is a consecutive set of arcs $a_1, \ldots, a_k$ where the head of each a_i with $i < k$ goes over at the corresponding crossing while the tail of a_1 and the head of a_k go under. A maximal *underpass* has the same definition, swapping "over" and "under".

The first phase of our algorithm finds overpasses and underpasses and reroutes them to have fewer crossings. Such reroutings are called *pass moves*. The second phase constructs a cubic lattice embedding of the knot from the diagram by first using graph-drawing methods to construct a planar grid embedding of the underlying 4-valent embedded planar graph associated to the diagram and then computing a piecewise-constant height function that respects over/under information. The heights are chosen to minimize the *total variation* $\mathrm{TV}(f)$—the sum of absolute height changes between consecutive arcs—subject to the constraint that overarcs are higher than underarcs at each crossing. The total variation has a nice geometric interpretation: it is the integral (over z) of the number of times the curve crosses the horizontal plane at height z. This means that minimizing TV produces the "vertically simplest" presentation of the knot. This constrained optimization problem can be reformulated as a minimum cost flow problem on a directed graph derived from the diagram and solved in near-linear time in the number of crossings.

Surprisingly, the minimum total variation of a diagram turns out to have an attractive topological interpretation: for an n-crossing diagram D, the minimum possible value of TV is $2(n-k)$, where k is the smallest number of crossings one must virtualize to make D virtually alternating (Proposition 8). In particular, this energy is a combinatorial invariant of the diagram. The difference from $2n$ is a measure of how far the diagram is from being alternating; for nonalternating diagrams, the optimal height function separates independent regions of the diagram from each other, often putting them on distinct levels. Reprojecting this lattice embedding from a different viewpoint produces a new diagram, often revealing new pass-move simplifications invisible in the original projection.

It is particularly interesting to try to simplify diagrams of the unknot, as a simplification algorithm which was guaranteed to produce a minimal (i.e., 0) crossing diagram would be a solution to the unknot recognition problem (see [14] for a discussion of the problem). Any such algorithm using Reidemeister moves must sometimes involve adding crossings [15] and any such algorithm using pass moves must sometimes involve pass moves which do not decrease the number of crossings [16, 17], but we do not know if there is always a weakly decreasing sequence of pass moves. By contrast, it is known that for grid diagrams there is a simplifying sequence which is weakly monotonic in a certain measure of diagram complexity [18].

Many authors have compiled tables or databases of complicated diagrams of the unknot [2, 17, 19]. Some of these are extremely challenging for existing codes to untangle. Such diagrams are colloquially referred to as "hard" unknots. We provide a highly performant implementation in `C++` [20] which successfully simplifies all published examples. We also simplify some large collections of new examples. See Section 4 for our performance results and a detailed description of our test set, which is publicly available [21]. While we do not claim that `ReAPR` solves the unknot recognition problem, we note that it succeeds on every published hard-unknot example we are aware of; we would welcome examples on which it fails.

2. Pass Moves and Pass Reduction

The first step in `ReAPR` is to find and perform reducing pass moves on a given knot diagram until we reach a diagram where no more reducing pass moves exist. We do this with a greedy algorithm, so there is no guarantee that the pass-reduced diagram where we terminate is the best one for the knot. In this section, we describe our algorithm at a level sufficient for conceptual understanding. Those interested in the implementation may also want to consult Section A, which gives more details on the algorithm.

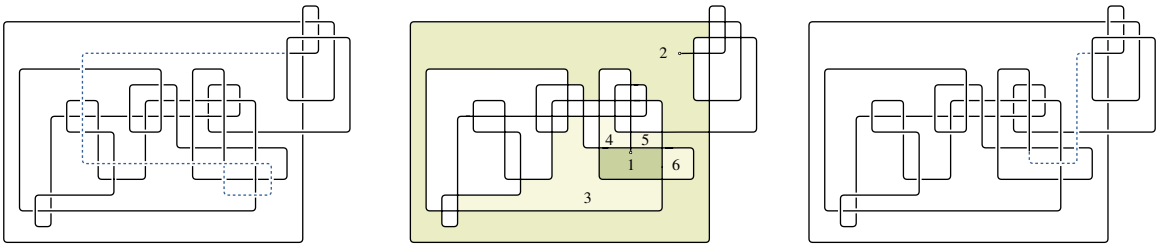


Figure 1: A simplifying pass move that is easy to discover in the pass graph, but hard to find by Reidemeister moves. The dotted overpass at left has 12 crossings. Bidirectional BFS in the dual graph visits the faces in the labeled order, finding the 3-crossing rerouting at right after visiting 6 faces (on the 7th step, we discover previously visited face 6 while exploring the frontier of face 2). Finding this rerouting by Reidemeister moves would require increasing the crossing count and conducting a large-radius search, as significant portions of the diagram separate the initial and final positions of the arcs.

We provide a `C++` implementation [20], noting that the performance results in Section 4 required very careful attention to memory layout and cache efficiency, which we also describe in Section A.

Suppose we find an over- or underpass $a_1, \ldots, a_k$ in a diagram D. We may imagine deleting it from the diagram, leaving a modified diagram $\hat{D}$ with two distinguished faces s and t consisting of the union of faces to the left and right of a_1 and a_k, respectively. In $\hat{D}$, the tail crossing of a_1 and the head crossing of a_k degenerate to two T-junctions. We could then reroute the pass by reconnecting these T-junctions along an arc through ℓ adjacent faces of $\hat{D}$, as in Fig. 1. Each face boundary crossed produces a new crossing, so if $\ell < k - 1$ the rerouting yields an isotopic diagram with fewer crossings. We call this a *reducing* pass move, and the graph of diagrams connected by pass moves the *pass graph*. The first part of our algorithm searches for reducing pass moves by finding shortest paths from s to t in the dual graph $\hat{D}^*$ (whose vertices are faces of $\hat{D}$ and whose edges connect adjacent faces), performing them in whatever order we find them.

We look for these shortest paths by searching $\hat{D}^*$ using bidirectional breadth-first search (BFS). A natural approach to computing shortest paths in $\hat{D}^*$ (used, for example, by `SnapPy`'s `Spherogram`) is to explicitly construct $\hat{D}^*$ in $O(n)$ time and search it with a standard graph library. However, the shortest path from s to t has length at most $k - 1$, so if $n \gg k$ this constructs far more of the dual graph than we will ever visit. We avoid this by navigating $\hat{D}^*$ implicitly using certain operations on directed arcs of D. In what follows, we refer to directed arcs as *darcs*.

Given a darc da in the original diagram D, we define operators reverse$(\cdot)$ and left$(\cdot)$ as follows. First, reversing the direction of da gives reverse(da). Next, if f is the face lying to the left of da, the *next left* darc left(da) is the next darc of f when cycling around the boundary of f in counter-clockwise order.

Note that faces of D (or vertices of D^*) can be identified with the orbits of the operator left$(\cdot)$, so we don't need to label them explicitly. Also, given a darc da on the boundary of a face f, the darc reverse(da) has to its left some face adjacent to f. So by cycling through the darcs of f and reversing each of them, we can visit neighbor faces of f. Since this visit of neighbors is the only kind of operation required for breadth-first search, we can search for shortest paths in D^* entirely using these two operations on darcs of D.

Of course, the goal is to search for shortest paths in $\hat{D}^*$, not in D^*. To do so, recall that $\hat{D}$ is

formed from D by deleting arcs. So to navigate in $\hat{D}^*$ it suffices to modify the left$(\cdot)$ operator by skipping over deleted arcs. This allows us to navigate $\hat{D}^*$ using only operations on darcs of D, without ever constructing $\hat{D}$ or $\hat{D}^*$ as separate data structures.

We now have all the tools needed for bidirectional BFS on $\hat{D}^*$. We perform this search using a *smallest-frontier* heuristic which we now explain. A *frontier* is a set of darcs pointing outward from the set of visited faces: da is in the frontier if the face to the left of reverse(da) has been visited but the face to the left of da has not. The initial frontiers $F(s)$ and $F(t)$ consist of the reverses of the darcs in faces s and t. To expand a frontier, say $F(s)$: for each $da \in F(s)$, we use left$(\cdot)$ to iterate around the face containing da, collecting all darcs encountered into a working set W. If any darc in W is in $F(t)$, the two searches have met and we reconstruct a shortest path by backtracking. Otherwise, the expanded frontier is a subset of $\{\text{reverse}(de) : de \in W\}$.

On a regular d-dimensional grid, strictly alternating bidirectional BFS visits a factor of 2^{1-d} fewer vertices than unidirectional search. However, the dual graph of a knot diagram is highly irregular: most faces are small, but a few (such as the exterior face of a polygon with many edges) can have tens of thousands of boundary arcs. Strictly alternating between the two frontiers risks expanding an enormous face early on one side, cascading into further large faces. We instead always expand the frontier with fewer darcs, a greedy strategy that avoids this blowup. In our benchmarks, bidirectional BFS with this smallest-frontier heuristic gave roughly a 20× improvement over unidirectional BFS on large diagrams.

2.1. Pass Moves vs. Reidemeister Moves

Why bother with pass moves at all? After all, both `SnapPy` and `Regina` have algorithms which navigate the Reidemeister graph either by breadth-first search or by random walk until a diagram with fewer crossings is found. Since every pass move is equivalent to a sequence of Reidemeister moves and every Reidemeister move is a pass move, the two graphs have the same connected components; in fact the Reidemeister graph is a subgraph of the pass graph. Therefore, these algorithms are equally powerful; either random walk or breadth-first search will eventually discover any reducing pass move. However, the geometry of the pass graph makes pass reduction much faster.

Suppose that D and D' are diagrams related by a reducing pass move which takes an overpass of length k to an overpass of length k'. It could take $O(n)$ time to discover the overpass, and then we might have to search the dual graph to radius $k' < k$ to discover the reducing edge in the pass graph. The dual graph is planar, so the number of faces examined is usually quite small.[1] At worst it is $O(n)$, as there are only $n + 2$ faces total. Once the edge is discovered, we can traverse it in $O(k)$ time, transforming D to D'. At worst, it will take $O(n^2)$ time to rule out the existence of any reducing pass move.

In the Reidemeister graph, discovering edges takes $O(n)$ time and traversing them is $O(1)$. This is much faster. However, to find D' by searching the Reidemeister graph, the search *radius* could be $O(n)$, as we might have to gradually push the overpass over an arbitrarily complex intermediate portion of the diagram as in Fig. 1 before we discover the rerouting on the other side. This could require us to generate and examine exponentially many diagrams. This is essentially why pass moves tend to be more efficient.

[1] However, there are link diagrams whose dual graphs contain $\sim k^\alpha$ faces in a ball of radius k for any fixed $\alpha > 1$ so it seems hard to bound this step usefully in terms of k. For examples, see Section 3.3 of [22], which constructs plane triangulations with this property. Subdividing each triangle into 3 quadrangles yields planar quadrangulations with the same property; every planar quadrangulation is the dual graph of a link diagram.

3. Re-embedding

Once pass reduction is finished and we have arrived at a diagram D with no reducing pass moves left, we turn to the re-embedding phase.

3.1. The Horizontal Component of Re-embedding

The first step is to construct an orthogonal layout for the embedded planar graph $G(D)$ using a new implementation of Tamassia and Tollis' algorithm [23] with the *turn-regularization* and *face saturation* procedures from [24] and with some internal modifications to improve performance and generate more attractive diagrams.

Finding an embedding of $G(D)$ can be done in linear time [25], but we invest somewhat more time in computing an embedding with a minimal number of bends, which requires solving several linear programming problems during layout. All of these are network flow problems on planar graphs [23]. Since our demands and costs are integral and bounded, these network flow problems may be solved in almost linear-time in the number of crossings ($O(n^{1+o(1)})$) [26]. In practice, we use the network simplex algorithm implemented in the class `MCFSimplex` from the library `Min-Cost-Flow-Class` project [27], which provides excellent performance and a clean `C++` interface. `SnapPy`'s `Spherogram` module and `Sage` provide Python implementations of comparable graph-drawing methods.

3.2. The Vertical Component of Re-embedding

Once we have the horizontal layout of $G(D)$, the next goal is to find a height function on over/undercrossings in the knot diagram D which satisfies the constraints imposed by the crossing information and which minimizes the total distance traveled in the z-direction while traversing the re-embedded knot.

Since the height function assigns values to over/undercrossings, let $v_0, \ldots, v_{2n-1}$ be the consecutive over/undercrossings as we traverse the knot diagram. If we define a height function $f(v_i)$ so that a_i is at z-height $f(v_i)$ when it leaves the tail crossing, then the over/undercrossing information would require $f(v_j) - f(v_i) \geq 1$ whenever a_j passes over a_i at a crossing. Since the arc a_i may need to change height before it arrives at its head crossing, the *jump* $|f(v_i) - f(v_{i+1})|$ is generally not zero. We can now formally define the total variation, which is the objective function we want f to minimize:

Definition 1. The *total variation* $\mathrm{TV}(f)$ is given by the total height of the jumps

$$\mathrm{TV}(f) = |f(v_1) - f(v_0)| + |f(v_2) - f(v_1)| + \cdots + |f(v_0) - f(v_{2n-1})|.$$

Therefore, we want to minimize $\mathrm{TV}(f)$ subject to the constraint that $f(v_j) - f(v_i) \geq 1$ whenever a_j crosses over a_i. To efficiently find a solution, we will write this as a minimum cost tension (MCT) problem and a corresponding maximum cost flow (MCF) problem.

To do so, we introduce an auxiliary directed graph $T(D)$ which is closely related to the chord diagram (or Gauss diagram) of D. The graph $T(D)$ will be an augmentation of the cycle graph with (ordered) vertex set $v_0, w_0, v_1, w_1, \ldots, v_{2n-1}, w_{2n-1}$. Conceptually, the added vertices w_i are the slack variables needed to express each absolute value $|f(v_i) - f(v_{i-1})|$ in $\mathrm{TV}(f)$ as the sum of $(f(v_i) - f(v_{i-1}))_+$ and $(f(v_{i-1}) - f(v_i))_+$.

Definition 2. The *tension graph* $T(D)$ of an n-crossing knot diagram D is a directed graph with $4n$ vertices and $5n$ edges. Let $v_0, \ldots, v_{2n-1}$ and $w_0, \ldots, w_{2n-1}$ be the $4n$ vertices. The edges are

in the form

$$\begin{aligned} v_i &\to w_i, \quad i \in \{0, \dots, 2n-1\} \\ v_{i+1} &\to w_i, \quad i \in \{0, \dots, 2n-2\}, \quad (\text{and } v_0 \to w_{2n-1}), \\ v_i &\to v_j, \quad \text{if arcs } a_i \text{ and } a_j \text{ are outgoing from a crossing where } a_j \text{ passes over } a_i. \end{aligned}$$

The edges $v_i \to v_j$ representing the crossings form a perfect matching between the vertices v_i, so we call them *matching edges* in $T(D)$. We call the other edges *cycle edges* as, if they were undirected, they would form a $4n$ vertex cycle graph with the v_i and w_i.

We now restate our problem as a constrained optimization problem on the tension graph $T(D)$:

$$\min_f \mathrm{TV}(f) \text{ subject to } f(v_j) - f(v_i) \geq 1 \text{ for each matching edge } v_i \to v_j. \tag{1}$$

We identify functions on the $4n$ vertices of $T(D)$ with vectors in $\mathbb{R}^{4n}$ and functions on the $5n$ edges of $T(D)$ with vectors in $\mathbb{R}^{5n}$. Moreover, we define the cost function c and the lower constraint function b by:

$$c(e) \coloneqq \begin{cases} 0 & \text{if } e \text{ is a matching edge,} \\ 1 & \text{if } e \text{ is a cycle edge,} \end{cases} \qquad b(e) \coloneqq \begin{cases} 1 & \text{if } e \text{ is a matching edge,} \\ 0 & \text{if } e \text{ is a cycle edge.} \end{cases} \tag{2}$$

Then we consider the minimum cost tension problem

$$\begin{aligned} \min_{f \in \mathbb{R}^{4n}} \sum_{e \in E} c(e)\,(f(\mathrm{head}(e)) - f(\mathrm{tail}(e))) \\ \text{subject to } f(\mathrm{head}(e)) - f(\mathrm{tail}(e)) \geq b(e) \text{ for all } e \in E. \end{aligned} \tag{3}$$

Lemma 3. *If f is an optimal solution to* (3)*, then $f(w_i) = \max(f(v_i), f(v_{i+1}))$ and therefore the sum over $e \in \{v_i \to w_i, v_{i+1} \to w_i\}$ of $c(e)\,(f(\mathrm{head}(e)) - f(\mathrm{tail}(e)))$ is given by $|f(v_i) - f(v_{i+1})|$.*

Proof. We know $f(w_i) - f(v_i) \geq b(v_i \to w_i) = 0$ and $f(w_i) - f(v_{i+1}) \geq b(v_{i+1} \to w_i) = 0$. Therefore, $f(w_i) \geq f(v_i)$ and $f(w_i) \geq f(v_{i+1})$, so $f(w_i) \geq \max(f(v_i), f(v_{i+1}))$. But if $f(w_i) > \max(f(v_i), f(v_{i+1}))$ we could reduce the overall cost of f (and maintain feasibility) by lowering $f(w_i)$ to $\max(f(v_i), f(v_{i+1}))$. It follows immediately that

$$c(v_i \to w_i)\,(f(w_i) - f(v_i)) + c(v_{i+1} \to w_i)\,(f(w_i) - f(v_{i+1})) = |f(v_i) - f(v_{i+1})|. \qquad \square$$

Lemma 4. *If $f \in \mathbb{R}^{4n}$ is an optimal solution of* (3)*, then the cost of f is equal to $\mathrm{TV}(f)$. Further, $f(\mathrm{head}(e)) - f(\mathrm{tail}(e)) \geq b(e)$ for all edges e of $T(D)$. Therefore, solving* (3) *is equivalent to solving* (1)*.*

Proof. By Lemma 3, the total cost of the cycle edges is $\sum_i |f(v_i) - f(v_{i+1})| = \mathrm{TV}(f)$. Since $c(e) = 0$ for all matching edges, this is the total cost of f. Further, for each matching edge, $b(v_i \to v_j) = 1$ so $f(v_j) - f(v_i) \geq 1$, as required. Conversely, any feasible solution f of (1) extends to a feasible solution of (3) with cost $\mathrm{TV}(f)$ by setting $f(w_i) \coloneqq \max(f(v_i), f(v_{i+1}))$. So the two problems have the same minimum value, and restricting an optimal solution of (3) to the v_i solves (1). $\square$

Example. A valid height function f is given by

$$f(v_i) := \begin{cases} 0 & \text{if } a_i \text{ is outgoing as an underarc,} \\ 1 & \text{if } a_i \text{ is outgoing as an overarc.} \end{cases} \tag{4}$$

This height function is too flat to make good embeddings for `ReAPR`, as projections of these embeddings to new diagrams tend to simply recover the original diagram. We will see below that if D is alternating, no improvement is possible. However, if D is nonalternating, we may reduce TV further. To see this, we now transform the MCT problem to its dual MCF problem. Again, we identify functions on the $5n$ edges of $T(D)$ with vectors in $\mathbb{R}^{5n}$.

Proposition 5. *The minimum cost tension problem* (6) *is the linear programming dual of the following maximum cost flow problem where $g \in \mathbb{R}^{5n}$ is a vector of (non-negative) flows on edges:*

$$\max_{\substack{g \in \mathbb{R}^{5n} \\ e \text{ a matching edge}}} \sum g(e), \text{ subject to } \begin{cases} g(e) \geq 0 & \text{for all } e, \\ \sum\limits_{\{e \mid v = \text{head}(e)\}} g(e) - \sum\limits_{\{e \mid v = \text{tail}(e)\}} g(e) + s(v) = 0 & \text{for all } v. \end{cases} \tag{5}$$

where $s(v) = +2$ if v is one of the v_i and $s(v) = -2$ if v is one of the w_i. That is, each v_i is a supply vertex with supply 2 and each w_i is a demand vertex with demand 2.

Proof. Let A be the $5n \times 4n$ incidence matrix[2] of the directed graph $T(D)$. That is, $A_{e,\text{head}(e)} = 1$, $A_{e,\text{tail}(e)} = -1$ for each edge in $T(D)$, and all other entries are zero. With the cost and bound vectors c and b from (2), the minimum cost tension problem (3) can be written as follows:

$$\min_{f \in \mathbb{R}^{4n}} c^\mathsf{T} A\, f \quad \text{subject to} \quad A\, f \geq b. \tag{6}$$

The LP dual is

$$\max_{g \in \mathbb{R}^{5n}} b^\mathsf{T} g \quad \text{subject to} \quad A^\mathsf{T} g = A^\mathsf{T} c \quad \text{and} \quad g \geq 0. \tag{7}$$

We interpret g as a flow on $T(D)$: the constraint $A^\mathsf{T} g = A^\mathsf{T} c$ requires net flow $(A^\mathsf{T} c)_v$ into each vertex v. Since each v_i is the tail of exactly two cycle edges and the head of none, $(A^\mathsf{T} c)_{v_i} = -2$; it follows that v_i is a *supply*. Likewise the inflow $(A^\mathsf{T} c)_{w_i} = +2$, so w_i is a *demand*. The objective $b^\mathsf{T} g$ counts only flow on matching edges. This is the claimed maximum cost flow problem. □

The formulation in terms of an MCT problem and its dual allows us to draw some conclusions about optimal solutions.

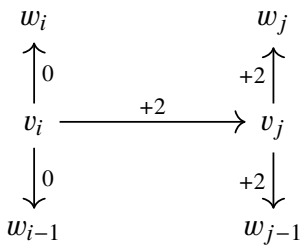
Proposition 6. *In an n-crossing diagram D, any height function f solving* (1) *has* $\text{TV}(f) \leq 2n$. *Moreover, D is alternating if and only if* $\text{TV}(f) = 2n$.

Proof. The example height function (4) is a feasible solution to the convex programming problem (1); therefore an optimal solution must exist. This solution has $|f(v_i) - f(v_{i-1})| \leq 1$ for all i, so $\text{TV}(f) \leq 2n$. In fact, $\text{TV}(f)$ is equal to the number of times a_i switches between being an overarc and underarc as we traverse the diagram, so it is $< 2n$ if D is nonalternating and equal to $2n$ if D is alternating.

[2]This matrix represents the discrete exterior derivative on the directed graph $T(D)$.

This last shows that an optimal solution f to (6) for an alternating diagram D has objective function $c^T A f \leq 2n$. We now demonstrate a feasible solution g for (7) with objective function value $b^T g = 2n$. This tells us that an optimal solution has $b^T g \geq 2n$. By strong duality, the objective functions for the dual problems (6) and (5) have the same value at an optimal solution; so this will suffice to show that this optimal value is $2n$. But this is also the total variation of the solution to (1) by Lemma 4.

Here is our feasible solution. For each matching edge $v_i \to v_j$, let us assign flows



Since the $v_i \to v_j$ are a perfect matching, this assigns flows to all edges. Further, it clearly obeys the conservation constraints at v_i and v_j. To see that it obeys the conservation constraints at all other vertices, note that since D is alternating the v_i alternate between being heads and tails of their matching edges as we proceed around the cycle. Therefore, each w_k is adjacent to exactly one v_l which has incoming flow from its matching edge. The corresponding $v_l \to w_k$ edge supplies flow 2 (exactly what we need to balance the demand at w_k) while the other incoming edge for w_k carries no flow at all. □

We can now describe the solution to the minimization problem quite explicitly.

Lemma 7. *There exists an optimal solution to* (5) *where each edge carries flow of* 0 *or* 2.

Proof. The constraint matrix of (5) is the incidence matrix of the directed graph $T(D)$, which is totally unimodular (this goes back to Poincaré, see [28]). By the theorem of Hoffman and Kruskal [29], a linear program with totally unimodular constraint matrix and integral right-hand side attains its optimum, if it has one, at an integral point. Our right-hand side is not merely integral but even, so consider the flow problem obtained from (5) by halving all supplies and demands to ± 1. Scaling flows by 2 is a bijection between the feasible sets of the halved and original problems which doubles the objective, so it matches optimal solutions to optimal solutions. The halved problem is feasible (route flow $\frac{1}{2}$ along every cycle edge and none along the matching edges) and its objective is bounded, so by Hoffman–Kruskal it has an integral optimal solution g'. Then $g := 2g'$ is an optimal solution of (5) in which every flow value is an even integer. Finally, every flow value of g is at most 2: each cycle edge carries at most 2 because the flows into its head w_i are non-negative and sum to exactly 2, and each matching edge carries at most 2 by conservation at its tail. Hence each flow value is 0 or 2. □

This gives us the following characterization of the minimal total variation energy of a diagram, which is rather knot-theoretic in flavor, but is most easily expressed in terms of virtual knot theory. Classically, each crossing in a knot diagram comes with over and under information. Virtual knot theory adds a third type of crossing, the "virtual crossing", which is neither over nor under. The "virtualization" of a crossing replaces an ordinary crossing by a virtual crossing. A virtually alternating diagram is one where the ordinary crossings of the knot alternate between

over and under, but the knot may also pass through any number of virtual crossings between ordinary crossings. An example alternating virtualization of the nonalternating knot 8_{19} appears in Fig. 2.

Proposition 8. *Suppose that D is an n-crossing diagram and k is the smallest number for which we can convert D into a virtually alternating diagram by virtualizing k crossings. Any optimal solution to* (1) *has* $\mathrm{TV}(f) = 2(n-k)$. *There is a corresponding height function with no more than $n-k$ maxima and $n-k$ minima.*

Proof. Call the matching edges which carry flow 2 in the optimal solution to (5) constructed in Lemma 7 the *flow edges*. Suppose there are m such edges and consider their $2m$ incident vertices $v_{i_1}, \ldots, v_{i_{2m}}$, which we assume to be in order among the cycle edges (recall Definition 2 and also see Figure 2). We label each of the $v_{i_1}, \ldots, v_{i_{2m}}$ as "incoming" if they are the head of their (unique) flow edge and "outgoing" if they are the tail of their (unique) flow edge. Our goal is to show that the incoming and outgoing v_{i_j} alternate as we go around the cycle edges.

We prove this by contradiction. With respect to the cyclic order on this list, suppose that two consecutive vertices have incoming flow edges. Without loss of generality, assume they are v_{i_1} and v_{i_2}. First, we claim that the edges $v_{i_1} \to w_{i_1}$ and $v_{i_2} \to w_{i_2-1}$ both carry flow +2. Since v_{i_1} receives 2 along its flow edge and has supply 2, its two outgoing cycle edges carry a total of 4; each carries 0 or 2 by Lemma 7, so both carry exactly 2, and likewise for v_{i_2}. Thus, they deliver +4 to the portion of the cycle between them: $w_{i_1}, v_{i_1+1}, \ldots, v_{i_2-1}, w_{i_2-1}$. On the other hand, this region can only absorb +2, since there is one more w vertex than v vertices. This is a contradiction. (Remember, we have assumed that no matching edges carry flow into or out of this region.)

A similar argument shows that adjacent vertices in this set cannot both have outgoing flow edges. It follows that the incoming and outgoing flow edges alternate, as desired. Since the direction of flow goes from underarc to overarc, this means that if we follow the diagram D around, ignoring crossings which correspond to matching edges with no flow, we must alternate between overarcs and underarcs. Of course, this is the definition of a virtually alternating diagram.

However, we may also observe that if we choose any collection of m matching edges with this alternation property, we may construct a corresponding solution to the flow problem by routing flow 2 across these matching edges (and distributing flow on the cycle edges accordingly). Therefore, the optimal solution to (5) must be given by a maximum size such subset of crossings of D. But that is the complement of a minimum size subset of crossings we need to virtualize. It follows that $m = n - k$, the optimal value of (5) is $2(n-k)$, and by strong duality together with Lemma 4, every optimal solution of (1) has $\mathrm{TV}(f) = 2(n-k)$.

We can learn more about the structure of the optimal height function f from the optimal flow function g using complementary slackness. Suppose that v_{i_j} is incoming, and consider the cycle edges between v_{i_j} and the (outgoing) $v_{i_{j+1}}$,

$$v_{i_j} \to w_{i_j} \leftarrow v_{i_j+1} \to \cdots \leftarrow v_{i_{j+1}}.$$

The edges $v_{i_j} \to w_{i_j}, v_{i_j+1} \to w_{i_j+1}, \ldots, v_{i_{j+1}-1} \to w_{i_{j+1}-1}$ all carry nonzero flow (in fact, they all carry flow +2). By complementary slackness, a nonzero dual variable implies an exactly satisfied primal constraint, so we know that the corresponding primal inequalities are actually equalities:

$$f(v_{i_j}) = f(w_{i_j}), f(v_{i_j+1}) = f(w_{i_j+1}), \ldots, f(v_{i_{j+1}-1}) = f(w_{i_{j+1}-1}).$$

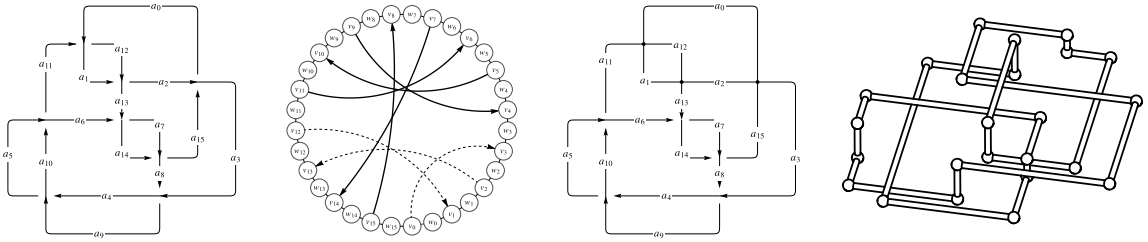


Figure 2: On the left, a nonalternating diagram with $n = 8$ crossings. This is the standard diagram for the 8_{19} knot. We have labeled the arcs a_i. In the center, we see the corresponding tension graph $T(D)$, with vertices v_i corresponding to the arcs a_i with vertices w_i inserted between them. The $k = 3$ matching edges which do not carry flow are dashed, while the $n - k = 5$ matching edges which carry flow 2 are drawn in black. The maximum cost is therefore 10. In the third panel, we see that if we virtualize the 3 crossings corresponding to the dashed edges, the new knot is virtually alternating (that is, it is alternating if we ignore virtual crossings). This must be a smallest set of crossings which makes D virtually alternating. On the far right, we see the corresponding lattice embedding, which has 3 local maximum arcs; this is less than the maximum of $n - k = 5$.

On the other hand, the interleaved cycle edges $w_{i_j} \leftarrow v_{i_j+1}, w_{i_j+1} \leftarrow v_{i_j+2}, \ldots, w_{i_{j+1}-1} \leftarrow v_{i_{j+1}}$ all carry zero flow. Here, we know only that the primal constraints apply (they may or may not have slack), leaving us with $f(w_{i_j}) \geq f(v_{i_j+1}), f(w_{i_j+1}) \geq f(v_{i_j+2}), \ldots, f(w_{i_{j+1}-1}) \geq f(v_{i_{j+1}})$. Combined, these inequalities tell us that $f(v_{i_j}) \geq f(v_{i_j+1}) \geq \cdots \geq f(v_{i_{j+1}})$, or that f is nonincreasing between v_{i_j} and $v_{i_{j+1}}$. Symmetrically, if v_{i_j} had been outgoing, we would have learned that f was nondecreasing between v_{i_j} and $v_{i_{j+1}}$. That is, an incoming v_{i_j} is a (weak) local maximum for f and an outgoing v_{i_j} is a (weak) local minimum. In particular, there are at most $n - k$ local minima and $n - k$ local maxima. □

Solving (1) as a generic ℓ^1 optimization problem with linear constraints using standard LP solvers was by far the slowest step in our initial implementation of the `ReAPR` algorithm. Rephrasing it as the MCF problem (5) was a very significant optimization. Using the network simplex algorithm from [27] reduces the computational cost to something comparable to the orthogonal graph layout.

4. Test set and Performance Comparisons

We tested our algorithm on approximately 2.6 million unknot diagrams with between 5 and 130,155 crossings, drawn from 10 test sets which we have deposited on DataDryad [21]:

(A) 200 random lattice unknots filling spheres, created by the BFACF [30–32] algorithm,
(B) the 18 unknots among the "children's game" knots of Gukov et al. [3],
(C) 200 self-avoiding unknotted 10,000-gons in tight spherical confinement [33],
(D) 5 complicated unknots of the "hard Goeritz" type from [15],
(E) the 21 "hard unknot" examples in [15],
(F) 27 examples of rational tangle unknots from "Kauffman's challenge" [34],
(G) 2,623,203 pass-reduced "hard unknots" found by Applebaum et al. [2],
(H) 600 unknotted random equilateral 3043-gons [35],
(I) 176 pass-reduced unknots found by Tuzun and Sikora [17],
(J) 384 pass-reduced "very hard unknots" from [2].

`ReAPR` recognizes all of these diagrams as unknots. We compared `ReAPR` to diagrammatic and non-diagrammatic unknot recognition algorithms in `Regina` [36] and `SnapPy` [37]. The diagrammatic simplification method `Link.simplify()` in `Regina` alternates between random Reidemeister III moves and simplifications using RI and RII moves. We also used a nondia-

			ReAPR		Regina				SnapPy			
set	knots	cr.			diagram		triang.		diagram		fund. grp.	
A	200	111K	**8.65s**	**1.0**	35h	.92	8.4h	.00†	55h	.00†	9.6h	**1.0**
B	18	18	.24	**1.0**	**.08**	**1.0**	8.42	**1.0**	4.30	**1.0**	3.90	**1.0**
C	200	18K	**1.08s**	**1.0**	18m	.92	79h	**1.0**	72h	**1.0**	14m	**1.0**
D	5	85	21.0	**1.0**	9.01	.20	212	**1.0**	1.5s	.60	**8.30**	**1.0**
E	21	45	25.0	**1.0**	5.71	.24	50m	.95	2.7s	.86	**7.20**	**1.0**
F	27	50	**0.99**	**1.0**	.80	.15	1.7h	.81	5.6s	**1.0**	2.9h	.89
G	≈ 2.623m	22	**17.56s**	**1.0**	51.3s	.00*	124h	1.0*	15h	1.0*	133.2s	**1.0**
H	600	3.3K	**167**	**1.0**	6.47s	.98	51m	**1.0**	20m	**1.0**	149s	**1.0**
I	176	15	38.0	**1.0**	**1.43**	**1.0**	166	**1.0**	259	**1.0**	44.0	**1.0**
J	384	28	**122**	**1.0**	8.51	.00	1.42s	**1.0**	23s	.02	175	**1.0**
	≈ 2.625m	33	**27.67s**	**1.0**	35h	.00*	215h	1.0*	143h	1.0*	13h	1.0*

Table 1: Comparison of unknot recognition methods across ten test sets (described on page 10; total: 2,624,834 diagrams). The "cr." column gives the average crossing number. Each entry shows total running time followed by effectiveness (fraction of unknots identified). Times are in milliseconds unless marked with "s", "m", or "h" (for "seconds," "minutes," or "hours," respectively). Effectiveness **1.0** = all unknots recognized; 1.0* rounds to 1.0; .00* rounds to 0. A dagger indicates that on all samples Regina or SnapPy exceeded time or memory constraints before terminating. Boldface marks the fastest method achieving perfect effectiveness.

grammatic method to recognize the unknot with Regina (this is "Regina triang." in Table 1):

```
L.simplifyToLocalMinimum()    (use RI and RII moves to simplify)
T = L.complement()    (build simplified triangulation)
T.simplify()    (run additional simplifications on triangulation)
check T.isSolidTorus()==true    (see if we recognized the unknot)
```

SnapPy's diagrammatic `Link.simplify('global')` method mixes pass moves and random RIII moves. We used the default limit of 100 random RIII moves. We used the SnapPy method `Link.deconnect_summand()` to split off connected summands where possible, which sometimes revealed further simplifications. We also used a non-diagrammatic unknot recognition method with SnapPy (this is "SnapPy fund. grp." in Table 1):

```
L.simplify('basic')    (use RI and RII moves to simplify)
E = L.exterior(with_hyperbolic_structure=false)    (triangulate complement)
G = E.fundamental_group(try_hard_to_shorten_relators=true)
check G.num_relators()==0    (see if we recognize the unknot)
```

All timings were obtained on a single core of a 2022 Mac Studio (Apple M1 Ultra). For Regina and SnapPy, we set a per-knot timeout of 100 seconds for the ≈ 2.6 million 'hard unknots' in set G, 1000 seconds for set A, and 3000 seconds for all other knots; no timeout was needed for ReAPR. The results are shown in Table 1.

ReAPR is the only method to achieve perfect effectiveness on every test set, processing all ≈ 2.6 million unknots in about 28 seconds of total CPU time. This is perhaps surprising: non-diagrammatic methods based on triangulation simplification [36] or fundamental group computation [37] are widely regarded as more powerful than any diagrammatic approach,

and the implementations in Regina and SnapPy are highly optimized. By contrast, ReAPR's underpinnings — pass moves, invented by Tait in the 19th century, graph drawing algorithms, and a minimum cost flow computation — are much more elementary. Nevertheless, it is competitive or faster on every test set.

The non-diagrammatic methods are indeed very strong. The SnapPy fundamental group simplifier beat ReAPR on test sets D and E of "classical hard unknots" and scored perfectly on every test set except F. The Regina triangulation simplifier was also generally effective, though there were 4,233 pass-reduced examples in G where it either timed out at 100 seconds or exceeded 4 GB of working memory. (These timeouts consumed about 95% of the runtime for test set G.) The two diagrammatic methods in Regina and SnapPy were less effective: Regina's `Link.simplify()` made no progress on test sets G and J (though it is extremely fast on small diagrams), while SnapPy's diagrammatic method failed on the larger pass-reduced examples in J.

Aside from ReAPR, all of these methods scale poorly to large diagrams. Both the Regina triangulation simplifier and the SnapPy diagrammatic method ran out of either time or memory on all members of test set A (200 lattice unknots, average $\approx$ 111K crossings); based on spot checks, we expect both of these methods would have eventually succeeded, but we project they would have taken ≈ 0.7 years and ≈ 1 year of CPU time, respectively. Regina's diagrammatic simplifier and SnapPy's fundamental group calculator were reasonably effective on test set A, but both took many hours. By contrast, ReAPR processes the same set in under 10 seconds. Scaling to large diagrams is a significant practical consideration: in [38], we needed to simplify diagrams with up to 35 million crossings.

The most striking result is test set F, the Kauffman challenge unknots [34]. These rational tangle unknots are the only test set where the non-diagrammatic methods struggle: even the SnapPy fundamental group simplifier fails on some examples, and both non-diagrammatic methods show unstable performance, sometimes recognizing a diagram quickly in one run and timing out at 3000 seconds on the next. The more complicated knots in F seem to be reliably difficult for these methods. Yet the same examples are straightforward for pass-move simplifiers: SnapPy `'global'` solves them in 5.6 seconds and ReAPR solves them in under 1 millisecond.

There are many other simplification strategies described in the literature. Petronio and Zanellati [39] use nonlocal moves which include pass moves and generalized versions of Reidemeister I and II moves which avoid other arcs crossing over or under a region to be simplified, as well as moves dividing diagrams into connect sums. Andreeva et al. [40] used an arc presentation to construct an interactive simplification applet in Java. Balch [41] used a combination of pass moves and some "hard" moves roughly similar to the Petronio–Zanellati moves. We did not test these methods because none of their implementations seem to be actively maintained.

Another very different approach uses the flow of a repulsive energy to unknot curves. These methods start with a geometric realization of the knot as a curve in space and use numerical optimization methods to try to minimize the energy. Here the state of the art is Noma et al. [42] which requires several seconds to untangle the 32 crossing Freedman–He–Wang unknot, faster than the method proposed by one of us (Schumacher) [43], but extremely slow compared to the tests above. These methods are often defeated by test set F (the Kauffman challenge unknots [34]). We did not test them here.

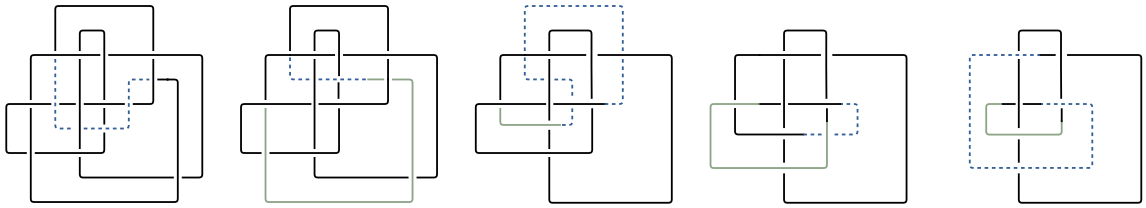


Figure 3: An illustration of "mid-level" crossing-number reducing pass moves in the 146th pass-reduced unknot discovered by Tuzun and Sikora [17]. The first two moves are of this type; after that, the diagram is reduced by conventional pass moves (the last few of which are left to the reader). `ReAPR` discovers mid-level pass moves empirically by providing a "side view" into the knot diagram.

5. Conclusions and Future Directions

The most pressing mathematical question about `ReAPR` is the simplest: why does it work? Petronio and Zanellati [39] noticed that even pass-reduced knot diagrams may yet contain crossing-number reducing passes which reroute "mid-level" arcs of a knot diagram, as in Fig. 3. `ReAPR` can certainly discover such moves by providing a new projection in which mid-level arcs become over- or underpasses. Preliminary experiments suggest that this accounts for most, but not all, of `ReAPR`'s simplifications: there appear to be cases requiring coordinated rearrangement of larger portions of the diagram. While `ReAPR` appears to appeal to 3d information about a lattice embedding, this embedding is computed from the diagram — in particular, the minimum total variation is a combinatorial invariant of the diagram (Proposition 8).

The effectiveness of pass moves at making large-scale changes to diagrams suggests that adding random crossing-number preserving pass moves might be a low-cost way to significantly improve the performance of diagram simplifiers (including `Knoodle` itself!). It also remains to be understood why the Kauffman challenge unknots are so difficult for the non-diagrammatic methods; perhaps there is something about the group presentations generated from these rational tangle diagrams which makes simplification particularly hard.

One important design goal of `ReAPR` was to be as fast and efficient as possible. As a result, `ReAPR` may be a useful tool in the training pipeline for AI methods in knot theory. For example, Applebaum et al. [2] conclude their paper with an example of a 42-crossing hard unknot which resisted 10^4 attempts to simplify it using `SnapPy`'s `'global'` heuristic; `ReAPR` simplifies this example in 0.36 ms on a 2025 laptop.

Acknowledgments

We are very grateful to our colleagues and friends for many helpful discussions about knots and unknots and their generous sharing of data, including Ronaldo Oliveira, Yo'av Rieck, Adam Sikora, and Lou Kauffman. Some of this work was done as part of the AIM SQuaRE "Landscapes of Knots", and we are grateful to the American Institute of Mathematics for their generous support. This work was partially supported by the National Science Foundation (DMS–2107700). H.S. gratefully acknowledges support through the research training group "Energy, Entropy, and Dissipative Dynamics", funded by Deutsche Forschungsgemeinschaft, project no. 320021702/GRK2326.

References

[1] Alan Turing. Solvable and unsolvable problems. In B. Jack Copeland, editor, *The Essential Turing*, pages 576–596. Oxford University Press, Oxford, September 2004. `doi:10.`

1093/oso/9780198250791.003.0024.

[2] Taylor Applebaum, Sam Blackwell, Alex Davies, Thomas Edlich, András Juhász, Marc Lackenby, Nenad Tomašev, and Daniel Zheng. The unknotting number, hard unknot diagrams, and reinforcement learning. *Experimental Mathematics*, 0(0):1–19, August 2025. doi:10.1080/10586458.2025.2542174.

[3] Sergei Gukov, James Halverson, Fabian Ruehle, and Piotr Sulkowski. Learning to unknot. *Machine Learning: Science and Technology*, 2(2):025035, June 2021. doi:10.1088/2632-2153/abe91f.

[4] Louis H. Kauffman, Nikolai E. Russkikh, and Iskander A. Taimanov. Rectangular knot diagrams classification with deep learning. *Journal of Knot Theory and its Ramifications*, 31(11):2250067, October 2022. doi:10.1142/S0218216522500675.

[5] Zhiyu Zhang, Yongjian Zhu, Yujie Zheng, Wenxuan Xu, and Liang Dai. Generating knotted polymer and protein structures by machine learning. *Communications Chemistry*, May 2026. doi:10.1038/s42004-026-02082-8.

[6] Yu Wang, Luwei Lu, Zhenhua Wang, Guoqiang Zhou, and Yuyuan Lu. Integrating graph convolutional network to BiLSTM for enhanced polymer knot identification. *Macromolecules*, 57(16):7980–7989, August 2024. doi:10.1021/acs.macromol.4c00931.

[7] Olafs Vandans, Kaiyuan Yang, Zhongtao Wu, and Liang Dai. Identifying knot types of polymer conformations by machine learning. *Physical Review E*, 101(2):22502, February 2020. doi:10.1103/PhysRevE.101.022502.

[8] Alexei Lisitsa, Mateo Salles, and Alexei Vernitski. Supervised learning for untangling braids. In *Proceedings of the 15th International Conference on Agents and Artificial Intelligence*, pages 784–789, Lisbon, Portugal, 2023. SCITEPRESS - Science and Technology Publications. doi:10.5220/0011775900003393.

[9] Audrey Lindsay and Fabian Ruehle. On the learnability of knot invariants: Representation, predictability, and neural similarity. *Advances in Theoretical and Mathematical Physics*, 30(1):191–209, 2026. doi:10.4310/ATMP.260413004720.

[10] Joseph Lahoud Sleiman, Filippo Conforto, Yair Augusto Gutierrez Fosado, and Davide Michieletto. Geometric learning of knot topology. *Soft Matter*, 20(1):71–78, 2024. doi:10.1039/D3SM01199B.

[11] Abdullah Khan, Alexei Vernitski, and Alexei Lisitsa. Reinforcement learning algorithms for the untangling of braids. *The International FLAIRS Conference Proceedings*, 35, May 2022. doi:10.32473/flairs.v35i.130657.

[12] Davide Castelvecchi. DeepMind's AI helps untangle the mathematics of knots. *Nature*, 600(7888):202–202, December 2021. doi:10.1038/d41586-021-03593-1.

[13] Anna Braghetto, Sumanta Kundu, Marco Baiesi, and Enzo Orlandini. Machine learning understands knotted polymers. *Macromolecules*, 56(7):2899–2909, April 2023. doi:10.1021/acs.macromol.2c02555.

[14] Marc Lackenby. Elementary knot theory. In *Lectures on Geometry*, Clay Lecture Notes, pages 29–64. Oxford University Press, Oxford, January 2017. doi:10.1093/oso/9780198784913.003.0002.

[15] Benjamin A. Burton, Hsien Chih Chang, Maarten Löffler, Clément Maria, Arnaud de Mesmay, Saul Schleimer, Eric Sedgwick, and Jonathan Spreer. Hard diagrams of the

unknot. *Experimental Mathematics*, 33(3):482–500, 2024. doi:10.1080/10586458.2022.2161676.

[16] Shuji Yamada. How to find knots with unit Jones polynomials. In M. Sakuma, editor, *Knot Theory: Proceedings of the Conference Dedicated to Professor Kunio Murasugi for His 70th Birthday (Toronto, July 1999)*, pages 355–361. Osaka University, 2000.

[17] Robert E. Tuzun and Adam S. Sikora. Verification of the Jones unknot conjecture up to 22 crossings. *Journal of Knot Theory and its Ramifications*, 27(3):1840009, 18, March 2018. doi:10.1142/S0218216518400096.

[18] Ivan A. Dynnikov. Arc-presentations of links: Monotonic simplification. *Fundamenta Mathematicae*, 190:29–76, 2006. doi:10.4064/fm190-0-3.

[19] Mitsuyuki Ochiai. Non-trivial projections of the trivial knot. In Claude Hayat-Legrand and Francis Sergeraert, editors, *Algorithmique, topologie et géométrie algébriques*, volume 192 of *Astérisque*, pages 7–10. Paris: Société Mathématique de France, 1990. URL: https://www.numdam.org/item/AST_1990__192__7_0/.

[20] Henrik Schumacher and Jason Cantarella. Knoodle: fast knot simplification and classification. https://github.com/HenrikSchumacher/Knoodle, 2025.

[21] Jason Cantarella, Henrik Schumacher, and Clayton Shonkwiler. Test sets for: Hard unknots are often easy from a different perspective, 2026. Dryad. doi:10.5061/dryad.vx0k6dk6v.

[22] Itai Benjamini and Oded Schramm. Recurrence of distributional limits of finite planar graphs. *Electronic Journal of Probability*, 6(23):1–13, January 2001. doi:10.1214/EJP.v6-96.

[23] Roberto Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM Journal on Computing*, 16:421–444, 1987. doi:10.1137/0216030.

[24] Stina S. Bridgeman, Giuseppe di Battista, Walter Didimo, Giuseppe Liotta, Roberto Tamassia, and Luca Vismara. Turn-regularity and optimal area drawings of orthogonal representations. *Computational Geometry*, 16(1):53–93, 2000. doi:10.1016/S0925-7721(99)00054-1.

[25] Roberto Tamassia and Ioannis G. Tollis. Planar grid embedding in linear time. *IEEE Transactions on Circuits and Systems*, 36(9):1230–1234, 1989. doi:10.1109/31.34669.

[26] Li Chen, Rasmus Kyng, Yang Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. *Journal of the ACM*, 72(3):19:1–19:103, 2025. doi:10.1145/3728631.

[27] Antonio Frangioni. Min-cost-flow-class. https://github.com/frangio68/Min-Cost-Flow-Class, 2017.

[28] Henri Poincaré. Second complément à l'Analysis Situs. *Proceedings of the London Mathematical Society*, s1-32(1):277–308, 1900. doi:10.1112/plms/s1-32.1.277.

[29] Alan J. Hoffman and Joseph B. Kruskal. Integral boundary points of convex polyhedra. In *Linear Inequalities and Related Systems*, number 38 in Annals of Mathematics Studies, pages 223–246. Princeton University Press, 1956. Reprinted in M. Jünger et al., editors, *50 Years of Integer Programming 1958–2008*, pages 49–76, Springer, Berlin, 2010; the DOI refers to the reprint. doi:10.1007/978-3-540-68279-0_3.

[30] Bernd A. Berg and Dietrich Foerster. Random paths and random surfaces on a dig-

ital computer. *Physics Letters B*, 106(4):323–326, November 1981. doi:10.1016/0370-2693(81)90545-1.

[31] Carlos Aragão De Carvalho and Sergio Caracciolo. A new Monte-Carlo approach to the critical properties of self-avoiding random walks. *Journal de Physique*, 44(3):323–331, 1983. doi:10.1051/jphys:01983004403032300.

[32] Carlos Aragão de Carvalho, Sergio Caracciolo, and Jörg Fröhlich. Polymers and $g|\varphi|^4$ theory in four dimensions. *Nuclear Physics B*, 215(2):209–248, 1983. doi:10.1016/0550-3213(83)90213-4.

[33] Ronaldo Oliveira. Personal communication, 2025. 200 self-avoiding unknotted 10,000-gons in tight spherical confinement.

[34] Louis H. Kauffman. From Knot Invariants to Knot Dynamics. In Renzo L. Ricca and Xin Liu, editors, *Knotted Fields*, volume 2344, pages 37–108. Springer Nature Switzerland, Cham, 2024. doi:10.1007/978-3-031-57985-1_2.

[35] Jason Cantarella, Henrik Schumacher, and Clayton Shonkwiler. A faster direct sampling algorithm for equilateral closed polygons and the probability of knotting. *Journal of Physics A: Mathematical and Theoretical*, 57(28):285205, 2024. doi:10.1088/1751-8121/ad54a8.

[36] Benjamin A. Burton, Ryan Budney, William Pettersson, et al. Regina: Software for low-dimensional topology. Available at http://regina-normal.github.io/ (28/04/2026), 1999–2026.

[37] Marc Culler, Nathan M. Dunfield, Matthias Goerner, and Jeffrey R. Weeks. SnapPy, a computer program for studying the geometry and topology of 3-manifolds. Available at http://snappy.computop.org (16/01/2025).

[38] Jason Cantarella, Tetsuo Deguchi, Henrik Schumacher, Clayton Shonkwiler, and Erica Uehara. Random knotting in very long off-lattice self-avoiding polygons. *Journal of Physics A: Mathematical and Theoretical*, 59(14):145205, 2026. doi:10.1088/1751-8121/ae55e3.

[39] Carlo Petronio and Adolfo Zanellati. Algorithmic simplification of knot diagrams: New moves and experiments. *Journal of Knot Theory and its Ramifications*, 25(10):1650059, September 2016. doi:10.1142/S0218216516500590.

[40] M. V. Andreeva, I. A. Dynnikov, and K. Polthier. A mathematical webservice for recognizing the unknot. In *Mathematical Software*, pages 201–207. World Scientific Publishing Co., July 2002. doi:10.1142/9789812777171_0020.

[41] Peter Balch. Unknotting a PCB. http://www.peterbalch.co.uk/UnknotPCB.html, 2016.

[42] Yuta Noma, Alec Jacobson, and Karan Singh. Medial sphere preconditioning for knot untangling and volume-filling curves. In *Proceedings of the SIGGRAPH Asia 2025 Conference Papers*, pages 1–10, Hong Kong Hong Kong, December 2025. ACM. doi:10.1145/3757377.3763856.

[43] Chris Yu, Henrik Schumacher, and Keenan Crane. Repulsive curves. *ACM Transactions on Graphics*, 40(2):10:1–10:21, June 2021. doi:10.1145/3439429.

[44] Boost C++ libraries. https://boost.org, 2026.

[45] Louis H Kauffman. State models and the Jones polynomial. *Topology*, 26(3):395–407, 1987. doi:10.1016/0040-9383(87)90009-7.

[46] Kunio Murasugi. Jones polynomials and classical conjectures in knot theory. *Topology*, 26(2):187–194, 1987. doi:10.1016/0040-9383(87)90058-9.

[47] Morwen B. Thistlethwaite. A spanning tree expansion of the Jones polynomial. *Topology*, 26(3):297–309, 1987. doi:10.1016/0040-9383(87)90003-6.

A. Pass Rerouting Details

A.1. Algorithmic Details

We provide here more details on the pass rerouting algorithm described in Section 2. Pass rerouting consists of three steps:

1. Finding a (maximal) over- or underpass.
2. Find a good target path to reroute to.
3. Performing the actual rerouting on the diagram.

We now explain each in some detail.

A.1.1. Finding a Maximal Pass

To find an over-/underpass in the link diagram D, we start at some active arc a_0. If its head goes over/under, we start searching an over-/underpass. For simplicity, let us assume that we are dealing with an overpass. (Underpasses are processed analogously.)

If its tail goes under, then we can be sure that this arc is the start of a maximal overpass. Otherwise, we are somewhere in the middle of a maximal overpass; then we walk backwards to its start. Once the first arc a_1 of the maximal overpass is found, we walk along the link, traversing arcs $a_1, a_2, \dots$ until we visit an arc a_k whose head goes under. Then $(a_1, \dots, a_k)$ is a (maximal) overpass. For later use we mark these arcs and all their crossings $c_0, \dots, c_k$. Since we have to reroute many passes, we do this in a way that allows us to unmark all arcs and crossings in an instant: For each crossing and each arc we set aside an integer for its mark and initialize these integers to 0. Whenever we start processing a new pass, we first increment an internal counter; as soon as we visit crossing c_i and arc a_i, we set their marks to the current value of this pass counter. This identifies them as members of this pass. (This way we also do not have to store the tuples $(a_1, \dots, a_k)$ and $(c_0, \dots, c_k)$ explicitly.) As soon as we start processing another pass, we increment the pass counter again, immediately rendering the marks on all arcs and crossings stale. So, in this sense, unmarking arcs and crossings costs $O(1)$.

A.1.2. Finding Shortest Paths

Let D be our current link diagram. To find a good target path to reroute to, we consider the new diagram $\hat{D}$ obtained from D by removing the marked arcs. Moreover, we let $\hat{D}^*$ be the *dual diagram* of $\hat{D}$: Its vertices are the faces of $\hat{D}$; its arcs correspond to the arcs of $\hat{D}$; and its faces correspond the crossings of $\hat{D}$ (but the latter do not really matter here).

Let $(a_1, \dots, a_k)$ be the pass we want to reroute. The two faces of a_1 in D collapse to one face f_1 in $\hat{D}$, which gives rise to a vertex in $\hat{D}^*$. Likewise, the two faces of a_k collapse to a face f_k in $\hat{D}$ and thus to a vertex in $\hat{D}^*$. Our goal is to find a shortest path of dual arcs $b_1^*, b_2^*, \dots, b_\ell^*$ in $\hat{D}^*$ from f_1 to f_k. Each of the dual arcs b_i^* means that we have to cross their corresponding primal arc b_i in $\hat{D}$, creating a new crossing. Such a rerouting allows us to replace the $k-1$ crossings

$c_1, \ldots, c_{k-1}$ of the overpass by just ℓ crossings, provided that this shortest path consists of $\ell < k - 1$ dual arcs. Hence, minimizing the path length in $\hat{D}^*$ means minimizing the number of crossings after rerouting, so what we are looking for is a shortest path in $\hat{D}^*$.

Creating a new knot diagram for $\hat{D}$ and creating a new dual diagram $\hat{D}^*$ costs $O(n)$. As explained in Section 2, we will instead navigate in $\hat{D}^*$ using the operators left$(\cdot)$ and reverse$(\cdot)$ on darcs.

The key difference in navigating on $\hat{D}^*$ rather than D^* is in modifying the left$(\cdot)$ operator to ignore deleted arcs, which we do using the marks on the primal arcs. Specifically, the only difference from traversing D^* is in what we do when we happen to visit a darc da whose underlying arc is marked as member of the current overpass. In this case, we simply ignore da and move on to $da \leftarrow \text{left}(\text{reverse}(da))$. This requires us to maintain a relationship between an arc a and its forward and backward darcs da_{forward} and da_{backward}. For example, one can set $da_{\text{forward}} = 2\,a + 1$ and $da_{\text{backward}} = 2\,a + 0$. This way, finding the arc underlying a darc amounts to a bit shift; the direction can be inferred by reading off the least significant bit; and computing reverse(da) amounts to flipping the least significant bit of the integer representing da. So the mapping between arcs and darcs can be done entirely without memory lookups.

The remainder of the pathfinding algorithm can be summarized as bidirectional Dijkstra search in $\hat{D}^*$ with unit edge weights on dual arcs. As the implementation details matter, we provide at least the most important ones. The idea of the bidirectional Dijkstra search is to grow two balls around the two starting vertices f_1 and f_k in $\hat{D}^*$. Let us call these balls A and B. We start with radii $r_A = 0$ and $r_B = 0$. Then one ball, say A, is selected, and its radius is increased by one unit. This is done by visiting the *boundary* of A and then visiting the neighbors of the boundary elements from there. If such a neighbor is already in A, we do nothing. If it is not in A we add it to A—unless it is already a member of the other ball B. In this case, the two balls A and B touch for the first time; then our search has come to an end. A shortest path can be reconstructed by maintaining for each visited vertex the vertex from which it was first visited.

Typically, this algorithm is described by focusing on the vertices. Since the vertices of interest are faces of the modified diagram $\hat{D}$ in our case, and since we do not even want to assign a name to them, we reformulate the usual bidirectional algorithm focusing on directed arcs of $\hat{D}$, which we identify with the subset of those darcs of D whose underlying arc is unmarked. For this we use two stacks A_{front} and B_{front} to maintain the lists of those darcs da whose duals "stick out" of A and B respectively. More precisely, a darc da is a member of A_{front} if and only if its right face (= the tail of da^*) is a member of A and if its left face (= the tip of da^*) belongs to neither A nor B. Moreover, we store the following information for each arc e of D:

- whether e has already been visited in the bidirectional search for the current overpass; this is done by storing the current value of the overpass counter;
- the arc underlying the darc that lead to e being visited;
- whether e was visited while growing A or while growing B;
- the direction in which e was crossed upon that visit ("left-to-right" or "right-to-left" with regard to the forward arc de_{forward}).

The first piece of information (the mark) allows us instant reset of information by the mechanism described above. The other data is used for the actual search and for the reconstruction of the optimal path through backtracking. At start-up of the search, the stacks A_{front} and B_{front} are empty and all arcs are implicitly marked as unvisited because their marks are stale. Then we cycle through the darcs of the faces f_0 and f_k (in the diagram $\hat{D}$) by the mechanism described above. For each darc da visited this way, we put the data described above onto the underlying

arc a, and push its *reversal* reverse(da) to A_{front} and B_{front}, respectively. Now the darcs on the stacks point to the faces we have to visit next for growing the balls A and B, i.e., these faces are to the left of these darcs.

Next we describe how to grow one of the balls, say A. First we provide an empty stack F to hold the new frontier of A. Then we pop a darc da from the stack A_{front}. Let us temporarily denote the face that lies to its left by f and the underlying arc by a. Now we use the cycling method described above to visit all the other darcs of the face f. For each darc de visited this way, we check whether and how the underlying arc e has been visited already:

- If e has not been visited from either A or B, we push reverse(de) to the stack F and store the following information in e:
  - the current value of the overpass counter (to remember that we have visited e);
  - that e was visited from a;
  - that e was visited while growing A; and
  - the direction in which e was crossed just now: "left-to-right" if $de = 2\,e + 1$ or "right-to-left" if $de = 2\,e + 0$.[3]
- If e has been visited while growing A in the same direction, then we stop cycling around f because we know that all other arcs have been visited, too.
- If e has been visited while growing A, but in the opposite direction, then we know that the face on the other side has been visited already. Hence, we do nothing and continue cycling over the darcs of f.
- If e has been visited while growing B, then the balls A and B just touch for the first time; we stop growing A and B immediately and move on to reconstructing the shortest path via backtracking.

We do this for all darcs on stack A_{front} until the latter is empty. Then we swap A_{front} with F so that A_{front} now represents the new frontier. (The stack F can be reused for the next frontier.)

All that is needed now is a strategy for deciding whether to grow A or B. Each such strategy leads to a different variant: always growing A leads to the original Dijkstra algorithm (in the special case of unit edge weights), also known as *unidirectional breadth-first search*; growing A and B in alternating fashion would resemble what is often referred to as *bidirectional breadth-first search*. The bidirectional search is more efficient because the main cost of growing a ball in a graph is to visit all the edges on the boundary.

However, as described in Section 2, we found in practice that another strategy works better: always grow the ball with the smaller frontier (or just the first one if the two frontiers have the same size). The size of the frontier is some indicator on how many edges are to be visited in the next sweep.[4] So, this is a greedy strategy to minimize the total number of edges (= dual arcs) visited.

A.1.3. Rerouting

This rerouting is somewhat tedious: The interior crossings $c_1, \ldots, c_{k-1}$ of the pass have to be disconnected from the arcs that do not belong to the pass. For each crossing disconnected this

[3] Note that this flag has to be assigned differently when we are about to grow B because the according part of the path will go backwards: here $de = 2\,e + 0$ means "left-to-right" and $de = 2\,e + 1$ means "right-to-left".

[4] The exact number depends on how many of the frontier's neighbor faces are unvisited and on the size of these unvisited faces. Alas, this information is not immediately available at this moment; so we have to use the size of the frontier as a proxy for this.

way, one of the incident arcs has to be deleted, and another has to be reconnected to where the other arc was connected—unless these arcs coincide; then the arc is deleted and an unlink is recorded. (This can happen only in diagrams for multiple-component links.) Let $b_1, \dots, b_\ell$ be the arcs crossed by the shortest path from f_1 to f_k. These arcs need to be split and a new crossing and two new arcs have to be sewn in appropriately. For the new crossings we can reuse the memory of crossings $c_1, \dots, c_\ell$ because we have the guarantee that $\ell < k - 1$; the remaining crossings are "deleted" by setting their state flags to "inactive" (see Section A.2.1 for details). Similarly, we can reuse the memory of the deleted arcs for the newly created arcs.

This all is straightforward, provided that neither the pass nor the target path contain any loops (except maybe $c_0 = c_k$). The target path cannot contain any loops because we chose it as a shortest path. But the pass may contain loops, in principle. Preventing this was the reason we put marks on the crossings during detection of passes (see Section A.1.1): once we revisit an already marked crossing, we know that the current strand forms a loop. We immediately stop expanding the pass and remove the loop by the surgery described above.

A.2. Implementation Details

A.2.1. Data Layout

Memory layout is crucial for the performance of our algorithm. The rerouting algorithm frequently alters the diagram. So it is tempting to design the data structure as a linked list: a crossing could be represented by a quadruple of pointers to the incident arcs; an arc could be represented by a pair of pointers to the crossings; and instances of crossings and arcs could be created (i.e., allocated) and deleted (i.e., deallocated) on the fly. This provides maximal flexibility, but it gives little control on where each crossing or arc is located in memory (which might lead to many expensive cache misses); and the overhead of frequent (de-)allocation may be considerable.

This is why we choose a different layout for the class `PlanarDiagram` in *Knoodle*. Rerouting always reduces the number of crossings and the number of arcs. Hence, we decided to make deletion fast and to entirely neglect creation of crossings and arcs. We store all n crossings in a 3-dimensional heap-allocated array `C_A` (read: crossings to arcs) of size $n \times 2 \times 2$ so that the index of an arc adjacent to crossing `c` can be accessed via `C_A[c][i][j]`, where `i` can be `0` (outgoing) or `1` (incoming), and `j` can be `0` (left) or `1` (right). In the layout convention of the *C* language this means that `C_A[c][0][0]`, `C_A[c][0][1]`, `C_A[c][1][0]`, `C_A[c][1][1]` are stored consecutively (and in this order). Likewise, we store the $m = 2n$ arcs in a 2-dimension array `A_C` (read: arcs to crossings) of size $m \times 2$ so that the indices of two crossing of arc `a` are located at `A_C[a][0]` (tail) and `A_C[a][1]` (tip). Since $4 = 2 \times 2$ and 2 are powers of 2, this allows the compiler to use fused add-with-shift instructions for indexing. Formally, we may treat `C_A[c]` as a 2×2-matrix and `A_C[a]` as a 2-vector.

Additionally, we store an n-vector of crossing state flags and an m-vector of arc state flags. The crossing state flags can take three states: "left-handed", "right-handed" or "inactive". The arc state flags can take only the values "active" or "inactive". At initialization, all crossing state flags are set to "left-handed" or "right-handed" according to the respective crossing's handedness, and all arc state flags are set to "active". Deletion of a crossing or an arc just amounts to setting the respective flag to "inactive"; the corresponding entries from the arrays `C_A` and `A_C` will be neglected from then on. These flags fit into a single byte. Storing them along with the 2×2-matrices and the 2-vectors for crossings/arcs would ruin their nice memory alignment. Moreover, these flags serve as guards against loading more information:

contemporary computer architectures always fetch contiguous blocks of bytes, the so-called *cache lines* at once. On the one hand, this allows to load and check many of these state flags before even a single arc or crossing has to be fetched at all. On the other hand, we can rely on the integrity of the data structure to navigate the active part of the diagram: if arc `a` is active, then the crossings `A_C[a][0]` and `A_C[a][1]` must be active, too, and we can jump to them without fetching and checking their state flags; and similarly for active crossings. Hence, it is advantageous to store the connectivity information and the flags separately. As the number of active crossings is crucial information, we maintain a counter for it that we decrement whenever we delete a crossing.

A.2.2. Ordering

The ordering in which we label the crossings and arcs and store them in `C_A` and `A_C` matters a lot; a bad ordering will incur a cache miss on always every memory access. A good one can avoid many of these misses, at least for a few frequently needed access patterns. Our two most important access patterns are:

(i) moving from one arc to its head crossing and then straight through the crossing to the opposite arc and so on; this is an access pattern very natural to knot theorists because it means "walking along the knot".
(ii) moving from one arc to its head or tail crossings and then turning *left*; we use this mode very frequent, e.g., to cycle around (connected components of) the boundary of a face and to navigate the dual graph during the Dijkstra search.

The default ordering we use is the "natural" ordering of a knot: arcs are numbered by the "time" of visit; crossings are ordered by the "time" of first visit. For multiple-component links we first traverse the link component of the first arc found in this order; then we move on to the link component of the next unvisited arc and so on. This is the perfect ordering for access pattern (i). We call this the *canonical ordering*. Our experiments show that this ordering is surprisingly good also for access pattern (ii): for large knot diagrams (e.g., obtained from random equilateral megagons), initializing with the canonical ordering instead of a random ordering can easily lead to a speedup of factor 5 or more for the rerouting code.

Of course, surgery and crossing/arc deletion change the ordering. Over time, this gradually reduces the cache-friendliness of our memory layout. We found that it is advantageous to *compress* our data structure from time to time, i.e., to reinitialize it with the canonical ordering, also removing all gaps that emerged from deletion of arcs and crossings. This requires traversal of the whole diagram, so it is expensive; but this cost amortizes quickly. To decide when to recompress, at a few decisive locations in our simplification pipeline we check whether more than half of the crossings are inactive; if they are, then we recompress. Adding this recompression strategy leads to a further speedup factor of about 1.2 for large diagrams. And for small diagrams it seems to do no harm.

A.2.3. The order in which over- and underpasses are processed

We process over- and underpasses in "rounds", interleaved with compression steps. At the start of each round, we remember the initial value of the pass counter. Then we look for the first active arc. If its head goes over/under, we start growing an over-/underpass. If its tail goes under/over in the opposite way, then we can be sure that this arc is the start of a maximal

over-/underpass. Otherwise, we are somewhere in the middle of a maximal over-/underpass and walk backwards to its start. Once an over-/underpass has been processed (by rerouting it or by deciding that rerouting is not salient), we immediately start a new under-/overpass at the last arc of the processed pass. That is, we walk backwards to find the start of a maximal pass containing this arc.[5]

Sometimes, we finish processing a pass whose last arc does not exist (because the whole pass was a loop) or whose last arc's follow-up arc has been recently rerouted. In that case, we instead look for the next active arc that has not been rerouted recently and continue the pass search there. If all arcs have been "touched" this way, we end the round, check to see if we should recompress it, and start a new round of rerouting. If a full round has been completed without any changes in the diagram, then we are done and stop.

A.2.4. Winged Half-edges

Two-dimensional diagrams are often encoded as winged half-edge data structures. For our use case, planar diagrams for knots and links, this would amount to storing for each darc da: its head crossing, its face to the left, the next left darc $\mathrm{left}(da)$, and the darc $\mathrm{right}(\mathrm{reverse}(da))$ to the right of $\mathrm{reverse}(da)$. (Moreover, one would select and store one of the adjacent darcs per crossing and per face.) Albeit this data structure is very useful for many mesh processing tasks (e.g., for mesh subdivision and remeshing), we found it not so well-suited for the kind of surgery we have to do in the rerouting phase: faces are split and merged quite frequently, so it is cumbersome to keep all darcs up to date.[6] Moreover, for the Dijkstra search on $\hat{D}^*$ we need neither know what the crossings are nor what $\mathrm{right}(\mathrm{reverse}(da))$ is. All we need to know is $\mathrm{left}(da)$; carrying along any further data, loading it into cache, and maintaining it would just slow us down.

We could compute $\mathrm{left}(da)$ on the fly by looking up the head crossing of da in `A_C` first and then looking up the left darc in `C_A`. However, these are two indirections, meaning two opportunities for cache misses. We can reduce this to a single lookup by precomputing an array `A_left_A` of size $2m = 4n$ whose da-th entry is $\mathrm{left}(da)$, and by indexing into that. The lookups in this array profit from a good ordering of arcs (which implies a similar ordering for darcs). This way, ordering of the crossings becomes irrelevant for the shortest path search.

The array `A_left_A` can be computed quickly by cycling over the active *crossings* of the diagram: each crossing can tell the four incoming darcs who their next left darc is (it's the reverses of the incoming darcs). Moreover, `A_left_A` is easy to update when any surgery is done. So this array needs to be computed only once for all pass moves (or when a recompression happens).

Since the shortest path search is memory bound, we indeed exhibit speedups close to a factor of 2 in practice (e.g., factor of about 1.8 for diagrams obtained from megagons).

A.2.5. Arrays vs. Hash Maps

Markers and backtracking information on arcs for the shortest path search can be stored in various ways. As we said before, we just used ordinary, contiguous, heap-allocated arrays of

[5] By walking backwards we make it likely that the accessed memory is still "warm", i.e., it still resides in a fast processor cache.

[6] This is why we decided to not track face information explicitly.

size m for this and uses a time stamp (the current value of the overpass count) for fast reset.

Another popular way to store information assigned to arcs is to use associative containers keyed on the arcs' labels. There are various implementations for such data structures. One way is to use a heap, i.e., a binary sort tree into which the key-value pairs are inserted as nodes. However, node insertion and node lookup cost $O(\log(k))$ on average, where k is the size of the heap. Moreover, depending on the implementation, deletion of the heap might cost $O(k)$ if the nodes need to be deallocated one by one. This is why we did not consider this data structure at all.

Another class of associative containers are hash maps; these offer insertions and lookups with cost $O(1)$. We tried the following C++ implementations: `std::unordered_map` from the *C++ standard library*; `boost::unordered_map`, and `boost::unordered_flat_map` of the *boost* library [44]. Of all these associative containers `boost::unordered_flat_map` performed best. However, the runtime for the pass rerouting was still three times as long as with our array implementation. This might come as a surprise at first: one might think that using an associative container is advantageous because the number k of visited arcs during the shortest path search should be relatively small compared to the number m of arcs. Alas, each of these containers have a cost of $O(k)$ for reset. And while insertions and lookups have constant cost, the cost for hashing cannot be neglected. Moreover, the very nature of hash maps may lead to a suboptimal memory access pattern. In contrast, our array-based implementation may profit from cache-friendly orderings of the arcs (and from repeated recompression).

A.2.6. Flagging Proven Minimal Diagrams

Once we have run the routine that detects and disconnects connected summands, we know that the diagram (and all its diagram components) are *reduced*: they do not contain any loops or isthmii. Afterwards, the routine that splits the diagram into separate instances of `PlanarDiagram` can check if the newly created diagrams are alternating. If yes, then the new diagram is minimal [45–47], i.e., its number of crossings coincides with the minimal crossing number of the underlying link class. In this case we set a flag in `PlanarDiagram` that excludes this diagram from any further rerouting attempts.