Framing holes within a loop hierarchy

Paulo E. Santos[*] Pedro Cabalar[†]

November 13, 2015

Abstract

We investigate the relation between non-trivial spatial concepts such as holes and string loops from a qualitative spatial reasoning perspective. In particular, we concentrate on a family of puzzles dealing with this kind of objects and explain how a loop formed in a string shows a similar behaviour to a hole in an object, at least regarding the qualitative constraints it imposes on the solution of the puzzle. Unlike regular holes, however, we describe how string loops can be dynamically created and destroyed depending on the actions on the string. Furthermore, under a Knowledge Representation point of view, we provide a formalisation that allows the different puzzle states to be described in terms of string crossings and loops, together with the actions that can be executed for a state transition and the complex effects they cause on the state representation. This implies the consideration of a formal representation of the side effects of actions that create or destroy string loops and the soundness of this representation with respect to the more general representation of string states in knot theory.

1 Introduction

The field of knowledge representation (KR) [29] in Artificial Intelligence aims at the explicit formalisation of domain knowledge in order to facilitate efficient reasoning. Within KR, the nature of spatio-temporal reasoning is *essential*, since almost all the other well-established KR and reasoning topics eventually show a spatio-temporal component, especially when they are put into practice in some application domain. As an example, think of the problem of putting on trousers and belt. One must pass one's legs through the trousers sleeves, button up (which means passing the button through the buttonhole), fasten the zipper, pass the belt through the belt loops in the trousers, and finally fasten the belt (which means passing the belt's tip through the belt buckle and the buckle bolt through a hole in the belt). All these operations involve actions on complex geometric figures (e.g. the pants, the zipper) and physical objects that involve different kinds of constraints, not only geometric and measure-related (such as choosing the right hole in the belt, depending on your waist size) but also about other features such as rigidity (the belt buckle) versus flexibility (the clothes and the belt). A

[*]Electrical Eng. Dep., Centro Universitrio da FEI, SP, Brazil
[†]Computing Dep., Corunna University, Spain

child learns how to dress partly by imitation and partly by her parents' explanations. A computer, in its turn, can nowadays create a very realistic 3D scene showing a child dressing up, using for that purpose a rich geometric model of the objects, but its "understanding" of the actions and movements performed, the constraints involved or the eventual goal is limited, if not non-existent in most cases[1]. This is also the case with other everyday spatial "puzzles", such as tying a shoe lace or netting, or with more specialised tasks, such as suturing or structured cabling. Moreover, if just rendering is a highly demanding computational task, an automated tool that could solve basic reasoning tasks involving flexible and rigid natural objects, such as simulation or planning, directly on a mathematical 3D model, simultaneously dealing with geometric and physical constraints while involving flexible objects, is unfeasible in practice.

Commonsense reasoning rarely deals with numeric temporal models or full geometric descriptions of space, but with *qualitative* descriptions instead. As a result, most work in spatio-temporal reasoning has focused on *Qualitative Spatial Reasoning* (QSR) [27] [7] [14], a field that attempts the formalisation of spatial knowledge based on primitive relations defined over elementary spatial entities (combinations of spatial representation with temporal relations are also research issues in the field [12]). Our previous research has focused on the investigation of spatial puzzles and games (some of them well-known and publicly available) as the ones shown in Figure 1. The reason for focusing on puzzles is that they constitute a good test bed, as they offer a small number of objects requiring a minimum background knowledge about unrelated features, while they keep enough complexity to constitute a challenging problem of KR. Thus, puzzles involving physical objects are our *Drosophila*[2], i.e. our base line from which we develop AI research. In particular, we have concentrated our efforts in formalising problems that involve flexible objects and holes, as they are very common in different scenarios like our commonsense example for putting on a pair of trousers. Most of these puzzles consist in releasing a rigid ring from an entanglement of strings and other rings or holed objects, and they are known to contain the basic features of solving elementary mathematics problems [13].

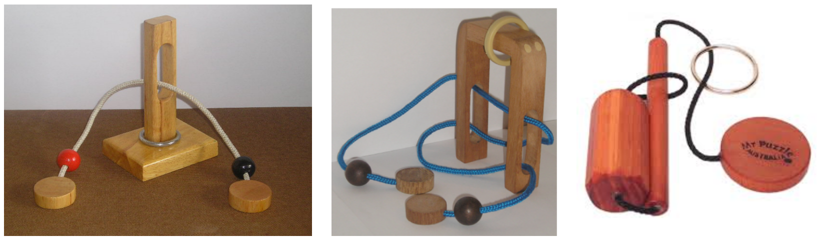

(a) Fisherman's Folly (b) Rope Ladder (c) Tricky Dick

Figure 1: Some puzzles for releasing a ring entangled with strings and holed objects.

Our methodology, applied through a series of papers [2, 20, 21, 22, 3, 23, 25, 24],

[1]The introduction of physical constraints in solid modeling constitutes, in fact, a challenging research area. See, for instance, the biannual ACM Symposium on Solid and Physical Modeling.

[2]Following Alexander Kronrod's metaphor about chess in AI, used in [26] and [15].

has consisted in a bottom-up strategy, starting from a very restrictive set of constraints and gradually relaxing them to cover puzzles with more challenging features. For instance, initial efforts [2, 20, 21] were put into solving the so-called Fisherman's Folly puzzle shown in Figure 1(a) using a list-based representation of string crossings. In [22] we also studied the spatial entities involved in the puzzle. All this work eventually led to an extensive paper [3] containing a complete logical formalisation in terms of Situation Calculus [17] and Equilibrium Logic [18] (an NMR approach generalising the stable model semantics for logic programs [9]). The formalisation in [3] allowed the representation of other puzzles such as the so-called Rope Ladder in Figure 1(b), and the *Tricky Dick* puzzle (Figure 1(c)[3]).

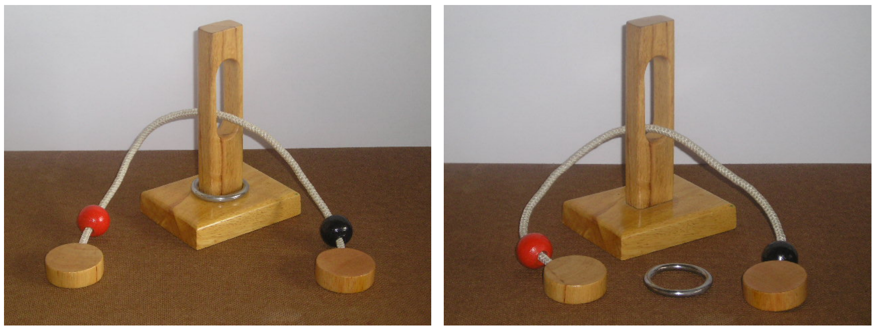
In this past work, however, several important limitations were overlooked in order to focus on a family of puzzles with common features. Among them, we had disregarded the representation of knots, the formation of string loops, and omitted the intermediate steps where a holed object was crossing another hole. The present paper concentrates on the definition of "loops", we outline and identify the main representational problems derived from the explicit consideration of string loops in the domains containing flexible objects (exemplified by the puzzles considered previously), and propose a possible formal solution that allows automated reasoning with strings, loops and holes in an analogous way.

2 Previous Formalisations

Previous work [2, 3, 22] has concentrated on the formalisation and automated solution of a number of spatial puzzles, the most basic of them was the so called Fisherman's Folly puzzle (shown in Figure 2). The goal of this puzzle is to release a ring from an entanglement of objects (maintaining the object's physical integrity). The elements of the Fisherman's Folly puzzle are a holed post ($Post$) fixed to a wooden base ($Base$), a string (Str), a ring ($Ring$), a pair of spheres ($Sphere1, Sphere2$) and a pair of disks ($Disk1, Disk2$). The spheres can be moved along the string, whereas the disks are fixed at each string tip. The string passes through the post's hole in a way that one sphere and one disk remain on each side of the post. It is worth pointing out that the spheres are larger than the post's hole, therefore the string cannot be separated from the post without cutting either the post, or the string, or destroying one of the spheres. The disks and the ring, in contrast, can pass through the post's hole.

In the initial state (shown in Figure 2(a)) the ring sits on the base, encircling the bottom of the post, and is supported on the post's base. The goal of this puzzle is to find a sequence of (non-destructive) transformations that, when applied on the domain objects, frees the ring from the other objects, regardless of their final configuration. Figure 2(b) shows one possible goal state. It is worth noting that this puzzle has four holes: the post hole, the ring hole and the two sphere holes. In the formalisation presented below, "holes" will be identified with their host objects, inspired by the general use of these terms in natural language (e.g. we usually say that "the post passes through

[3]Figure 1(c) was adapted from `http://www.mrpuzzle.com.au/tricky-dick-wood-and-rope-puzzle.html`



(a) Initial (b) Goal: the ring must be free

Figure 2: A spatial puzzle: the Fisherman's Folly.

the ring", rather than "the post passes through the ring's hole"). An ontology of holes was proposed in [5, 6], and applied to the domains considered in this paper in [21].

The simple solution for the Fisherman's Folly puzzle presented in [2] relies on distinguishing the three sorts of objects: *holes* (which includes the post hole, the ring hole and the holes through the spheres), long objects (that includes the string and the post), and regular objects (including all the remainder objects). For each hole h, its faces are distinguished: h^- and h^+; and for each long object l its tips l_b and l_e are defined (where b and e respectively stand for "begin" and "end").

A puzzle state can be schematised by diagrams such as the one in Figure 3, which shows the initial state of Fisherman's Folly (cf. Figure 2(a)). Arrows correspond to segments of long objects, defined between pairs of hole crossings, or between a hole crossing and a tip. These arrows point in the direction from tip l_b to tip l_e of a same long object l. Ellipses represent holes and boxes are linked regular objects. The positive face of a hole is determined by a right-hand screw rule from the small arrow in the ellipse (similar to a spin direction). The orientation of the holes are assigned such that all the holes in the initial state of a puzzle have the same orientation (i.e., for instance, all the holes have a counterclockwise orientation at the initial state).

For a simpler visual recognition of hole faces, we have emphasised the ellipses as if they were solid surfaces and the string crossings as little "cuts" on them so that the "visible side" corresponds to the positive hole face. Notice how a hole could also be seen as a long object closed on itself joining its two tips.

Central to this simple solution is the definition of a list data structure named $chain(X)$. This data structure represents the sequence of all hole crossings on a long object X, when traversing X from its beginning tip (X_b) to its ending one (X_e). For instance, the state shown in Figure 3 is represented by the following two chains: $chain(Post) = [Ring^+]$ and $chain(Str) = [Sphere1^+, PostH^+, Sphere2^+]$. The former represents that the long object $Post$ crosses the ring hole and the latter states that the string crosses first the hole on the sphere 1, then the post hole and, finally, the hole on the sphere 2, respectively. Note that, for brevity, only the outgoing hole faces are shown, following the string direction from begin to end.

Actions in this work will be arguments of the function $s_i = do(\alpha, s)$ that denotes

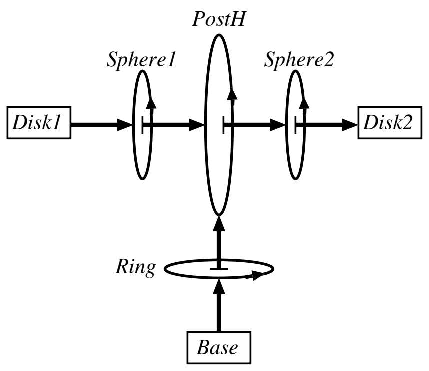


Figure 3: Schematic representation of the initial state [3].

the resulting state of applying an action term α to a previous state s. The constant S_0 denotes the *initial situation*.

An action $pass$ was formally defined in our previous work [3] to represent the movements of puzzle objects. The effects of $pass$ either add or delete a hole crossing from the $chain$ on which it is applied. Figure 4 shows the case of *passing* an ending tip (*End*) of a long object towards the p face of a hole (movement *passing end right* (PER)), and back (movement *passing end left* (PEL)), where the symbol *hi* represents any hole crossing in the chain. Passing a beginning tip (*Begin*) is shown in Figure 5, represented by the movements *passing begin right* (PBR) and *passing begin left* (PBL). Figures 6 and 7 show the cases related to the action of passing a hole h toward the p face of another hole: *pass hole right* (PHR1 and PHR2) and *pass hole left* (PHL1 and PHL2).

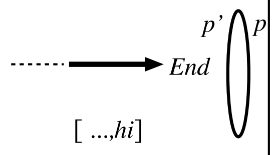
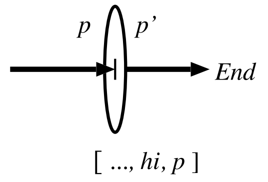
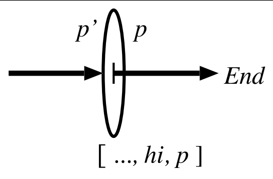
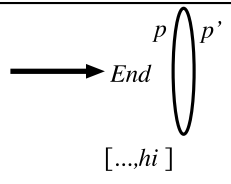
Situation	*Movement* (PER)	*Movement* (PEL)
s		
do(pass(End ,p),s)		

Figure 4: Movements for passing the ending tip $pass(End, p)$ [3].

Using these definitions, a solution to the Fisherman's Folly puzzle can be repre-

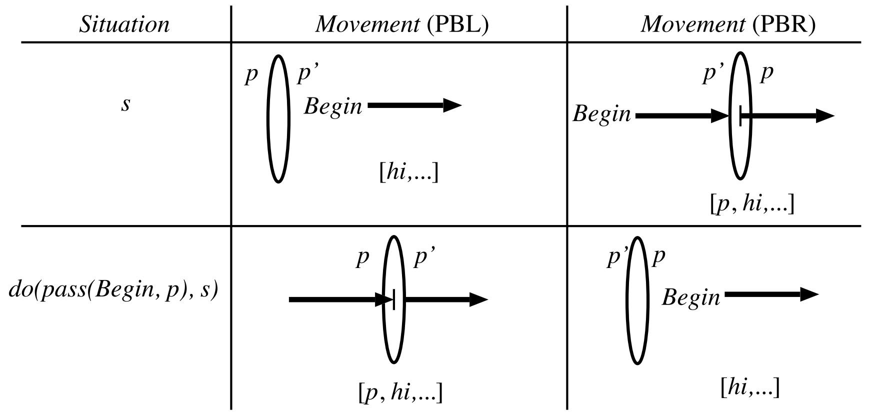


Figure 5: Movements for passing the beginning tip $pass(Begin, p)$ [3].

sented by the sequence of chains shown on Figure 8, whereby each state is identified by its sequence number plus the pair of lists $chain(Post)$ and $chain(Str)$ in this order. Note that state s_5 has actually reached the goal since, at this point, the ring hole $Ring$ does not occur in any list, i.e., it is not crossed by any long object[4].

The question that naturally follows is whether the representation and reasoning system defined in [3] can be extended to support a puzzle whose solution depends on the manipulation of loops, such as the easy-does-it puzzle, shown in Figure 9(a)[5]. An essential part of the easy-does-it solution is "passing a loop trough a ring" (e.g. the state change from the initial state of Easy-does-it, Figure 9(b), to the state shown in Figure 9(c), this action is passing the loop[6] $l(Str2, R3^+, [1, 2])$ through $R1$ towards its positive side). Therefore, this domain calls for the explicit representation of string loops.

The easy-does-it puzzle (Figure 9) is going to be used as an example throughout this paper. Thus, we introduce the chain description of the initial state (S_0) depicted in Fig. 9(b) as follows:

$$chain(P, S_0) = [P_b^-, R2^+, R1^+, P_e^+]. \tag{1}$$

$$chain(Str2, S_0) = [\ Str2_b^-, R3^+, R3^-, Str2_e^+\ ]. \tag{2}$$

$$chain(Str1, S_0) = [\ Str1_b^-, l(Str2, R3^+, [1, 2])^+, Str1_e^+\ ]. \tag{3}$$

A first observation is that each $chain(X, S)$ (for any state S) includes now the beginning and ending tips of the long object X; moreover, they are respectively signed

[4] This is assuming that when $Sphere2$ is passed through the ring it takes the string with it rather than leaving some of it looping through the ring. This assumption will be relaxed further in this paper, when we introduce a "pull" action.

[5]Figure 9(a) was adapted from `http://www.puzzlesolver.com/puzzle.php?id=78`

[6]The arguments related to the loop notation shall become clear further in this paper.

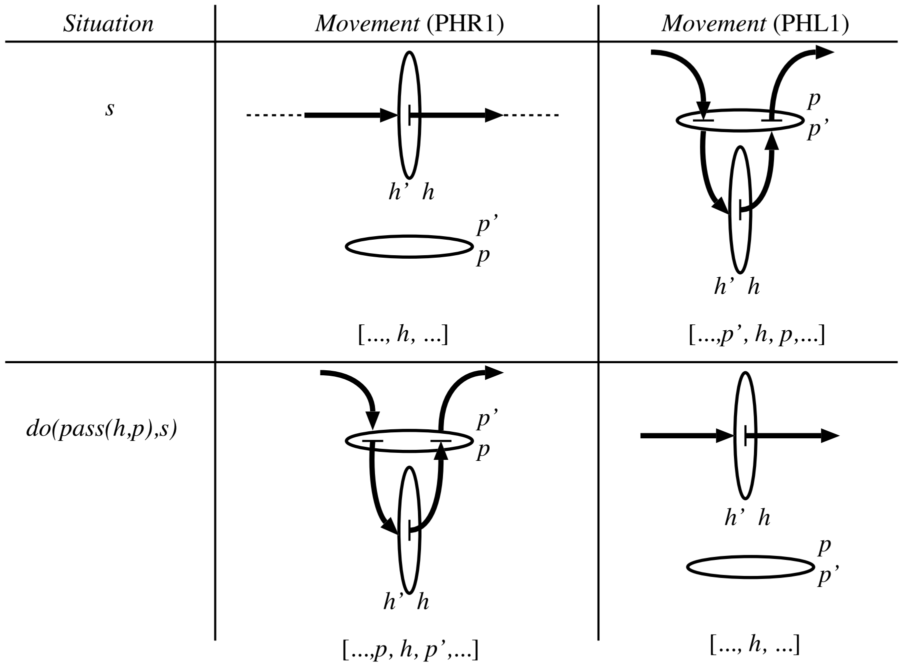


Figure 6: Movements (PHR1), (PHL1) for action $pass(h, p)$ [3].

as X_b^+ and X_e^- (this notation is explained in the next section). As a second observation, notice that the string $Str2$ forms a loop in its interaction with ring $R3$. This loop consists of (the segment between) the two crossings $\langle R3^+, R3^- \rangle$ occurring in Formula (2) and becomes a new hole denoted as $l(Str2, R3^+, [1, 2])$. This new hole is crossed, in its turn, by string $Str1$ as shown in its chain, described in Formula (3). In the next section we explain the meaning of this notation in details.

3 String Loops

As a first approach to the string-loop problem, in this paper we describe the puzzle domains by means of strings only. Restrictions due to other types of objects (e.g. rigid objects) will be considered as imposed domain constraints. Note that the puzzles cited in the previous sections could be constructed using strings only, with some added constraints (e.g. the string-only version of Fisherman's Folly would be the direct construction of the diagram in Figure 3, with added domain constraints expressing which objects do not pass to which holes).

In this work, strings are assumed to be infinitely extensible, ruling out the need to deal with metric information at this point.

We use the term *hole crossing* to refer to the point where a string crosses a hole and the term *string crossing* (or *string intersection*) to refer to the point of intersection

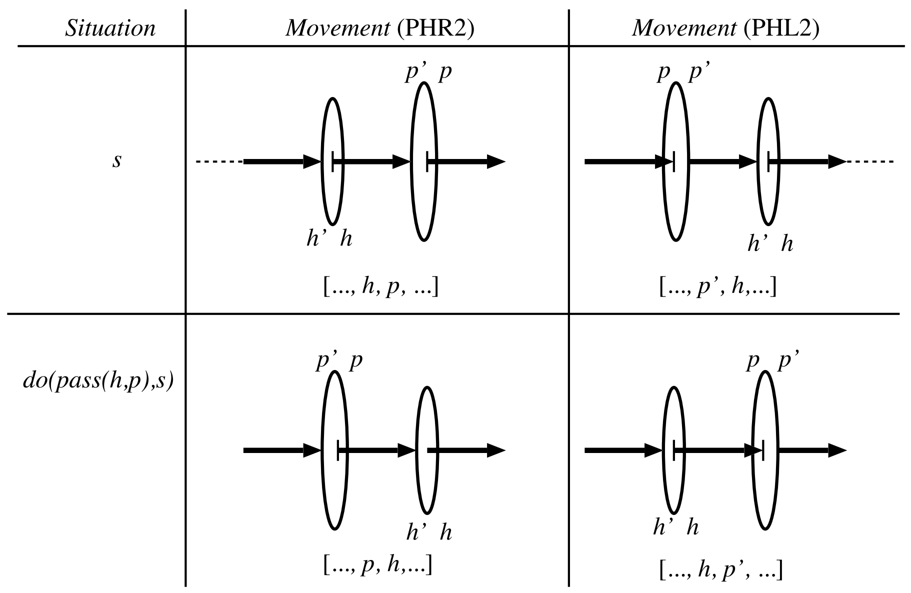


Figure 7: Movements (PHR2), (PHL2) for action $pass(h, p)$ [3].

between two strings (or between two distinct points of a single string). Whenever it is clear from the context, the more general term *crossing* is going to be used in either case.

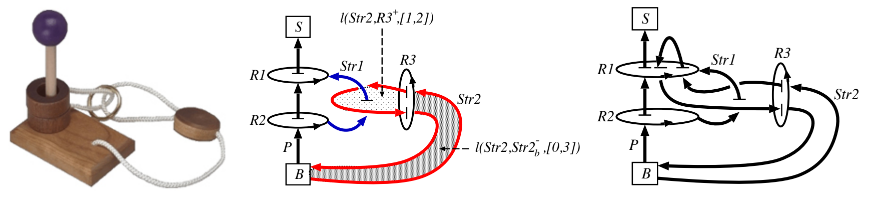
Loops in the string form holes that are represented with two faces '+' and '-' following our previous representation of holes in rigid objects [2, 3, 22], as introduced above. The only distinction here is that loops can appear and disappear when they are pulled through a hole.

This paper makes the following assumptions on the diagrammatic depiction of puzzles (some of them inherited from knot diagrams in knot theory [11]):

- all the rings in an initial state have the same orientation;
- all hole crossings are explicitly shown in the diagrams;
- all intersections are point intersections (i.e. they do not happen on a continuous section of the string);
- there are only single intersections (i.e. at each intersection point in the diagram, at most two lines meet);
- the tips of a string are not part of intersections.

state	$chain(Post)$	$chain(Str)$
S_0	$[Ring^+]$	$[Sphere1^+, PostH^+, Sphere2^+]$
$s_1 = do(pass(\{Str^+, Disk2\}, PostH^-), S_0)$	$[Ring^+]$	$[Sphere1^+, PostH^+, Sphere2^+, PostH^-]$
$s_2 = do(pass(\{PostH, Post^+\}, Ring^-), s_1)$	$[\,]$	$[Sphere1^+, Ring^-, PostH^+, Ring^+, Sphere2^+, Ring^-, PostH^-, Ring^+]$
$s_3 = do(pass(\{Sphere2\}, Ring^-), s_2)$	$[\,]$	$[Sphere1^+, Ring^-, PostH^+, Sphere2^+, PostH^-, Ring^+]$
$s_4 = do(pass(\{Ring\}, PostH^+), s_3)$	$[\,]$	$[Sphere1^+, PostH^+, Ring^-, Sphere2^+, Ring^+, PostH^-]$
$s_5 = do(pass(\{Sphere2\}, Ring^+), s_4)$	$[\,]$	$[Sphere1^+, PostH^+, Sphere2^+, PostH^-]$

Figure 8: A formal solution for the Fisherman's puzzle [2].



(a) Easy does it (b) Diagram - S_0 (c) Diagram - S_1

Figure 9: The easy-does-it puzzle.

3.1 Defining loops on chains

String loops (or simply "loops") in this work are considered to be created only by crossing the same string twice through a hole. In other words, this only deals with loops that are either created by *picking* a segment of a string through a hole, or that are the results of linking the two tips of a string. This is supported by the formal representation of loops below. We leave for a future work the representation of loops defined by crossing a string on itself, or by passing a tip of a string through the same hole twice in the same direction, as shown in Figure 10, where $R1$ is a ring and $S1$ is a string. These definitions call for a broader theory about knots [4, 19].

Given a string S, in this work we define a loop in S as any pair of crossings, in opposite directions, with respect to the same hole. We say that a loop $l = [i, j]$ on the hole h is on the positive ('+') side of h if and only if i has sign '+' and j has sign '-'. Loops on the '-' side of h are exactly the opposite: i has sign '-' and j has sign '+'.

We are going to use the term *endpoints* to represent hole crossings defining loops.

Let S be a state. A loop on a string St formed on a hole H is represented by means of the function constructor l (called *l-term*) with the following shape:

- $l(St, H^{sign}, [N_1, N_2])$.

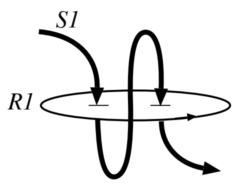


Figure 10: Passing a tip of a string through the same hole twice on the same direction.

The informal meaning of this notation is the following:

> a loop on string St formed by all segments between the endpoint position in the chain N_1 (that passes towards hole face H^{sign}) and the endpoint N_2 (that comes back towards the opposite face H^{-sign}).

Endpoints in a $chain(St, S)$ (e.g. $[N_1, N_2]$) are represented by natural numbers starting from 0 (for each chain). An l-term will only occur in a chain when another string is crossing the corresponding loop. For instance, as we saw in Formula (3) (resp. Figure 9(b)), $Str1$ crosses the loop $l(Str2, R3^+, [1, 2])$ towards its positive face. However, for representing a state, it is convenient to recognise all the existing loops, even though they are not crossed by any string, since the actions we will perform may make them relevant at a given moment. As an example, $Str2$ in Fig. 9(b) actually has a second, larger loop depicted in dark grey, that comprises all segments in $Str2$. This second loop (represented as $l(Str2, Str2_b^-, [0, 3])$) is interesting because it *contains* the previous loop, as shown in Formula 4 below:

$$chain(Str2, S) \quad = \quad [\overbrace{\underbrace{Str2_b^-}_{=B}, \quad \underbrace{R3^+, R3^-}_{l(Str2, R3^+, [1,2])} \quad , \underbrace{Str2_e^+}_{=B}}^{l(Str2, Str2_b^-, [0,3])}]. \tag{4}$$

Another interesting feature of this loop is that, rather than being a consequence of crossing another hole, it is formed because the two tips of $Str2$ are linked together to the puzzle base B, as we have emphasised in Formula 4 with the equalities $Str2_b^- = B$ and $Str2_e^+ = B$. As we can see, a string can also form a loop when its two tips are directly or indirectly linked. To cope with this situation in an homogeneous way, we may think that the tips Str_b and Str_e in a long object Str actually define a pair of *virtual holes* that constitute the initial and final crossings in $chain(Str, S)$. We take the criterion that Str_b is always crossed towards its negative face, and Str_e towards its positive one. For instance, the leftmost diagram in Figure 11 represents string Str free, where the corresponding chain would be $chain(Str, S) = [Str_b^-, Str_e^+]$. If we approach both tips until we join them (or they get joined to the same object) we reach the situation in the rightmost diagram where holes Str_b and Str_e become *the same*, i.e., $Str_b = Str_e$ and we use Str_b to denote both. In this case, the string has $chain(Str, S) = [Str_b^-, Str_b^+]$ and ends up forming a single loop $l(Str, Str_b^-, [0, 1])$. Therefore, in our Easy-does-it example, if we follow this convention, the larger loop of $Str2$ in Figure 9(b) can be denoted as $l(Str2, Str2_b^-, [0, 3])$.

If a closed string R is persistent, i.e., it cannot be made open by any domain action, it will receive the name "Ring" and we will abbreviate its loop $l(R, R_b^-, [0, 1])$ simply as the string name R by a slight abuse of notation. Rings are denoted by the capital letter R, assigned with an index i ($i \in \mathbb{N}$).

For brevity, the literals $sign$ and N_2 will be sometimes omitted in $l(S, H^{sign}, [N_1, N_2])$ whenever they are clear from the context. Similarly, if the tips of a string X_b, X_e are linked to the same object B, we will sometimes use that object name as initial and final crossings in the chain.

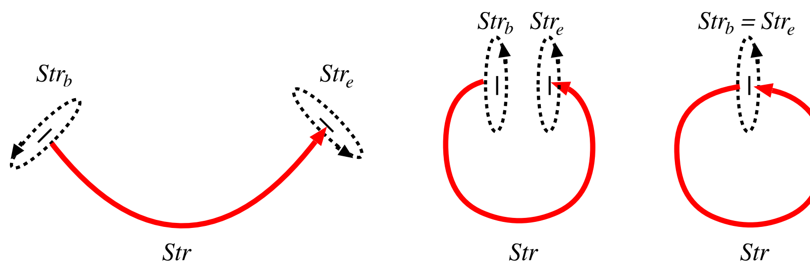
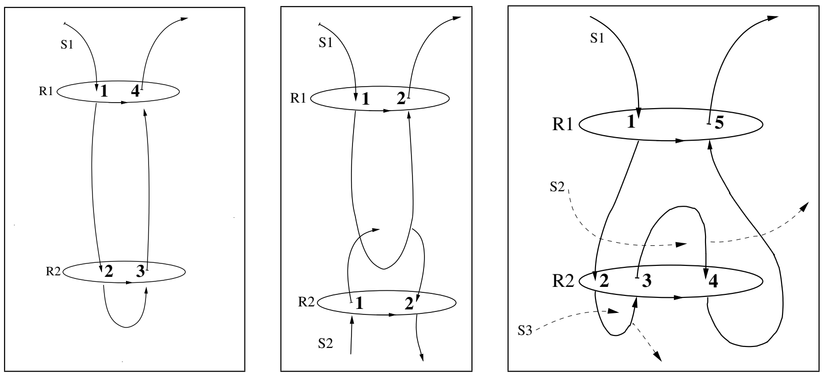


Figure 11: When both tips are joined, the whole string forms a loop.

These simple definitions of chains and loops can handle cases where a single string makes loops through various holes (Fig. 12(a)), or various strings that define loops and pass through loops made by other strings (Figs. 12(b) and 12(c)). The chains (and related loops) representing Figures 12(a), 12(b) and 12(c) are shown in Formulae 5, 6 and 7, respectively.



(a) $State_i$ (b) $State_j$ (c) $State_k$

Figure 12: String loops.

$$chain(S1, State_i) = [S1_b^-, \underbrace{R1^-, \overbrace{R2^-, R2^+}^{l(S1,R2^-,[2,3])}, R1^+}_{l(S1,R1^-,[1,4])}, S1_e^+]. \tag{5}$$

$$\begin{cases} chain(S1, State_j) = [S1_b^-, \underbrace{R1^-, l(S2, R2^-, [1,2])^-, R1^+}_{l(S1,R1^-,[1,2])}, S1_e^+], \\ chain(S2, State_j) = [S2_b^-, \underbrace{R2^-, l(S1, R1^-, [1,2])^-, R2^+}_{l(S2,R2^-,[1,2])}, S2_e^+]. \end{cases} \tag{6}$$

$$\begin{cases} chain(S1, State_k) = [S1_b^-, \underbrace{R1^-, \underbrace{R2^-, \rlap{$\overbrace{}^{l(S1,R2^+,[3,4])}$}R2^+}_{l(S1,R2^-,[2,3])}, R2^-, R1^+}_{l(S1,R1^-,[1,5])}\ S1_e^+], \\ chain(S2, State_k) = [S2_b^-, l(S1, R2^+, [3,4])^-, S2_e^+], \\ chain(S3, State_k) = [S3_b^-, l(S1, R2^-, [2,3])^-, S3_e^+]. \end{cases} \tag{7}$$

The idea described above, for calculating loops when repetitions of crossings are detected, has to be extended to handle cases where the loops are crossed by strings (such as in the case shown in Figure 12(c)). This issue is solved by defining a hierarchy of loops, as presented in the next section.

3.2 A hierarchy of loops

In order to introduce the hierarchy of loops, we first consider a hole h and m crossings (endpoints) through it from a single string (as shown in Figure 13) defining a sequence of loops, where each endpoint is marked with a natural number. For brevity, in this section, we identify each loop solely by its endpoints.

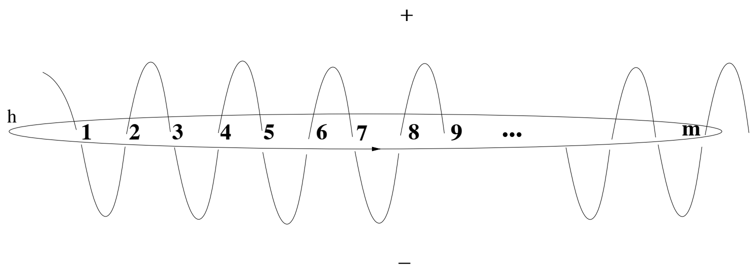


Figure 13: Various loops from one string through a single ring.

Using the definition from the previous section, in Figure 13 the pairs of endpoints:

$$\{[1,2],[2,3],\ldots,[i,i+1]\}$$

define loops, where the loops whose starting points are odd numbers are on the negative side, and those with even starting points are on the positive side of h. Note that there are also *implicit* loops in Figure 13, such as those defined by the pairs $[1,4],[1,6],\ldots,[1,m]$ (on the negative side of h), the pairs $[2,5],\ldots,[2,m-1]$ (on the positive side of h), and so on.

We use Allen's interval algebra [1] to qualify the possible relations between pairs of loops on a single string, where the loops' endpoints are represented within the endpoints of intervals. Briefly, Allen's interval algebra [1] is defined by a set of 13 jointly-exhaustive and pairwise-disjoint base relations representing the possible relations between pairs of intervals. Given two intervals, x and y, the Allen interval relations are described in Figure 14.

Applying Allen's relations to the loops on Figure 13, we can say, for instance, that the loop $[1,2]$ *starts* the loop $[1,4]$; that $[2,3]$ occurs *during* $[1,4]$ and that $[3,4]$ *finishes* $[1,4]$.

Let $l_1 = [i,j]$ and $l_2 = [k,w]$ be two loops on h defined on two distinct endpoints (i.e., either $i \neq k$ or $j \neq w$), we say that l_1 is an *inner* loop of l_2 ($l_1 \leq l_2$) iff $l_1[starts,\ during,\ finishes]\ l_2$, read as "$l_1$ either *starts* or *is during* or *finishes* l_2". If this is the case, we say that l_2 is an *outer* loop with respect to l_1 ($l_2 \geq l_1$). We call *elementary loop* a loop that has no associated inner loops (for instance, $[1,2]$ and $[2,3]$ are elementary loops), as defined below.

Definition 1. *l is an elementary loop if and only if there is no l' ($l' \neq l$) such that $l' \leq l$.*

The relation *inner* ($\leq$) (resp. *outer* ($\geq$)) forms a partial order on loops. We can say that inner loops and their related outer loops, that fall on the same side of a hole, form

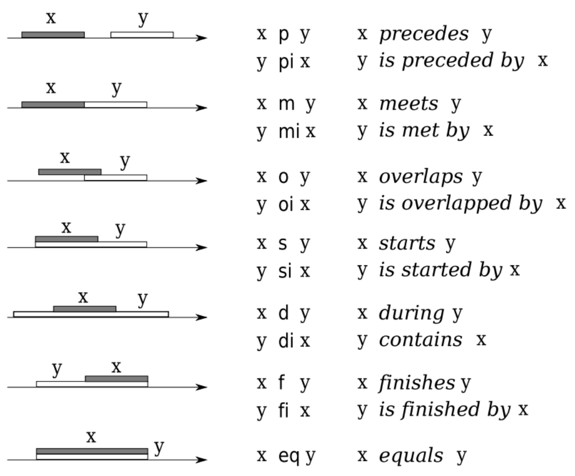


Figure 14: Allen's Interval Relations (adapted from [14]).

a hierarchy where crossings that occur on inner loops are inherited by their immediate outer loops. E.g., if there is a string S crossing a loop l_i on one side of h, and this loop is undone by an action (e.g. by *pulling* the loop through the hole that defines it), then this crossing will be assigned to the next outer loop with respect to l_i.

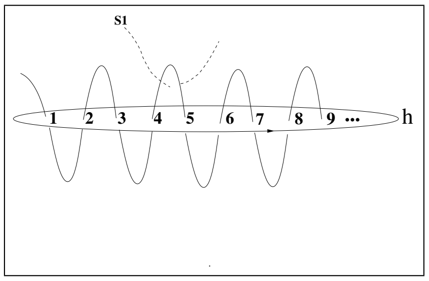
For instance, Figure 15(a) shows a string $S1$ initially crossing the loop $[4, 5]$ and Figure 15(b) represents the effect of *pulling* this loop through the hole. Although loop $[4, 5]$ is destroyed in this action, and string $S1$ now forms itself a loop through h, $S1$ is still crossing the loop $[2, 7]$ (note that if $[3, 6]$ is pulled, $S1$ will not be "free" as an effect, but will still be crossing $[2, 7]$). This is also the case with crossings on adjacent (inner) loops to the loop destroyed (as shown in Figure 16). Figure 16 shows the case of a string S1 previously crossing the loop $[1, 2]$ (Fig. 16(a)), this crossing is inherited by the outer loop $[1, 4]$ when $[2, 3]$ is pulled (cf. Fig. 16(b)). The effects of actions on loops will be formalised in Section 4 below.

It is worth noting that, when loops defined on the first or last endpoints of a sequence (e.g. on endpoints 1 or m in Fig. 13) are undone, there is no outer loop on the same ring that inherits their crossings. Thus, if there is no other loop (formed on a distinct ring) that inherits this crossing, the string is free. The case of loops formed from multiple rings is considered in Section 3.3.

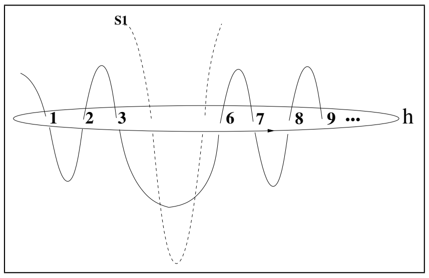
Figure 17 depicts the loop hierarchy, where the dotted columns represent the interval endpoints defined by the loops.

3.3 Generalising the loop hierarchy

The inheritance of crossings through loops on a single string, but defined on distinct holes, has to be explicitly handled only when actions are applied to the first or last endpoints on a loop sequence on a particular hole. These are the cases where the string crossings on loops defined on one hole may be inherited by a loop defined on another

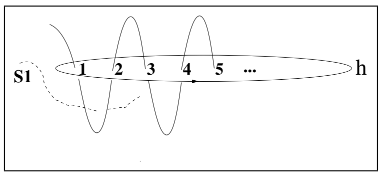


(a)

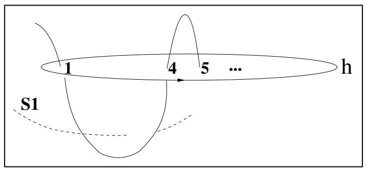


(b)

Figure 15: String loops.



(a)



(b)

Figure 16: String loops.

hole. For instance, Figure 18 shows two rings $R1$ and $R2$ and a string S passing, first, towards $R1^-$ (crossing at a), then crossing many times through $R2$ (endpoints 1 to m) and finally passing towards $R1^+$ (at endpoint b). Thus, the string S forms a loop through $R1$ ($[a, b]$) and, in between the endpoints of this loop, S also has loops defined on the endpoints 1 to m (on R_2). The loops starting at endpoints 2 to $m-2$ are handled as explained in the previous section. However, when loops defined on 1 and m are undone, their eventual crossings with other strings are inherited by the loop $[a, b]$.

More formally, the inheritance of crossings on a single hole can be extended to the case of multiple holes in the following way: an outer loop l on a chain inherits the crossings of any inner loops defined over any holes in the same chain, that have the same sign of l. For instance, in Figure 19(a) eventual crossings on the loop $[2, 3]$ are inherited by the loop $[1, 4]$, but the same does not happen with the analogous loops on Figure 19(b), since in this case $[1, 4]$ has a distinct sign from $[2, 3]$. This is also shown on the chain descriptions of Fig. 19(a) (Formula 5) and Fig. 19(b) (Formula 8). Note that loop $[2, 3]$ in Figure 19(b) falls in the same plane as loop $[1, 4]$. If it were the case that $[2, 3]$ resided on a distinct plane from $[1, 4]$ (i.e. if $[1, 4]$ was contained on the same plane as this paper and $[2, 3]$ was sticking out of the paper) a crossing on $[2, 3]$ could also cross $[1, 4]$ (or not). Thus, the 2D diagram would have to be drawn in a distinct way, making explicit all loop crossings (according to the guidelines for drawing string diagrams mentioned in Section 3).

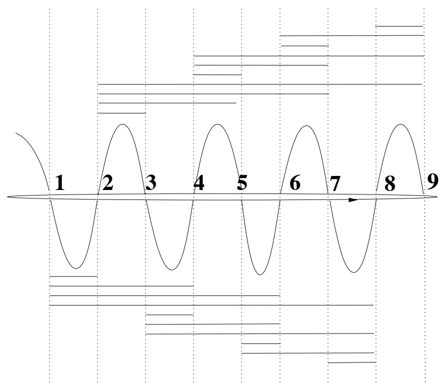


Figure 17: Hierarchy of loops for one string through a single ring.

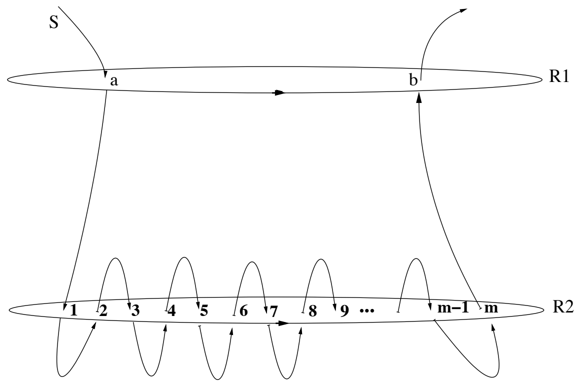


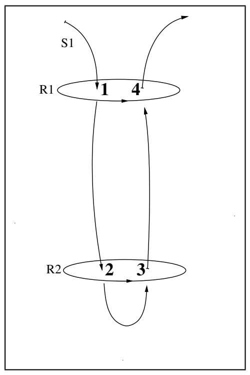
Figure 18: Loop hierarchy on multiple holes.

$$chain(S1, State_j) = [S1_b^-, \underbrace{R1^-, \overbrace{R2^+, R2^-}^{l(S1,R2^+,[2,3])}, R1^+}_{l(S,R1^-,[1,4])}, S1_e^+]. \quad (8)$$

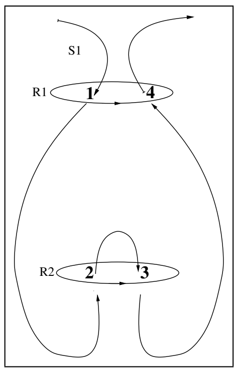
4 Actions on loops

This section defines actions on elementary loops. For this purpose, we introduce a formal definition for segments, as first proposed in [3].

As mentioned above, the set of hole *crossings* is the essential information used on the *chain* representation. Formally, a *crossing* is the overlapping region between a hole and a long object. A *segment* can be defined as a maximal continuous portion of a long object not overlapping with any hole, such that there cannot be a segment of the string x connecting more than two hole crossings. Each segment of a string x



(a) $State_i$



(b) $State_j$

Figure 19: Inheritance of crossings from distinct rings.

at a given state will be denoted as $x:i$ where $i \in \mathbb{N}$ is a *segment label* that uniquely identifies the segment. As we did with crossings in $chain(x, S)$ (for any state S), we also start numbering segments with 0. In this way, if we have $n + 1$ crossings (e.g., $chain(x, S) = [C_0, \ldots, C_n]$) then we have n segments, $(x:0)$,...,$(x:n-1)$, being each one $(x:j)$ delimited by crossings C_j and C_{j+1}.

Let us start by considering the idea of loop creation. Rather than passing an object to a hole side, we consider an action $pick(x : i, p)$ meaning that we pick some arbitrary segment i on a string x (that is not one of the string tips) pulling from it towards the hole side p. Figure 20 shows the result of this movement.

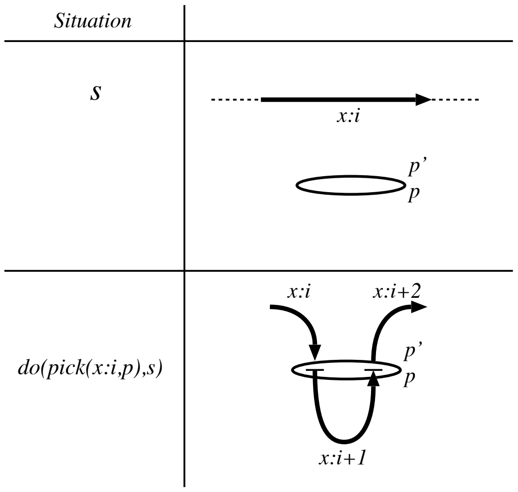


Figure 20: Creating a loop by picking a segment i inside string x.

In principle, the pick action can always be executed regardless of the origin and target of segment $x : i$. Notice that this action will always create a new loop. Thus, for instance, if in the resulting situation, depicted in Figure 20 (bottom), the action $pick(x : i+1, p')$ is performed (that is, the segment $i+1$ in x is *picked* toward the hole side p') the initial situation s is not obtained as a result, but the one shown in Figure 21 (third column) instead.

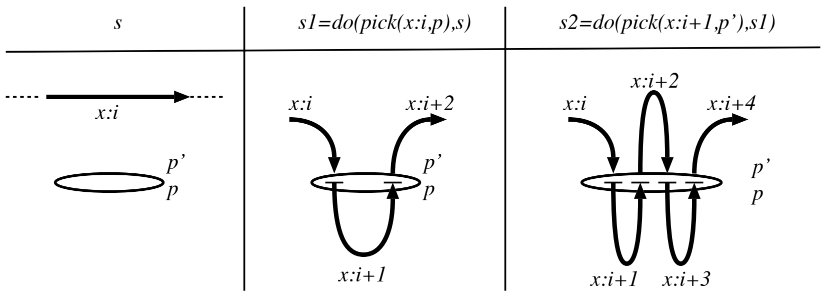


Figure 21: Picking back a point in a loop creates more loops.

The difficult case of a pick action happens when a segment of a loop (that happens to be crossed by many strings) is picked, creating inner loops (cf. Figure 22). The question is to which inner loops these crossings are assigned to as an effect of this action. In this work we assume that the crossings are assigned to the leftmost inner loop (as shown in Fig. 22, left) since, in most of the puzzle domains, assigning a specific inner loop to a string crossing is irrelevant to the solution. However, a more specific version of *pick* can be defined to identify the crossing with the exact inner loop that it should be assigned. Defining this enhanced pick action is outside the scope of this paper.

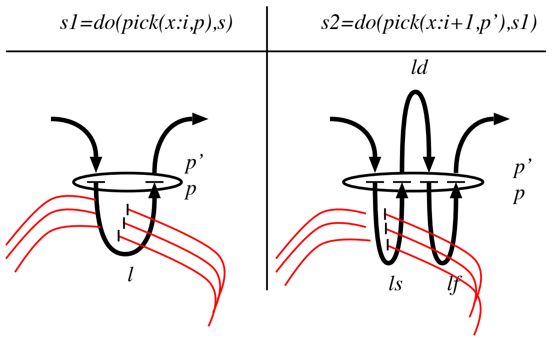


Figure 22: Picking back a point in a loop creates more loops.

The second movement related to loops considered in this work is the action of *removing a loop*, that we call $pull(l)$, where l is a loop. This action is the exact inverse of the $pick$ action defined above, whereby $pull(l)$ eliminates loop l completely, passing it back through the hole that defines it (as shown in Figure 23).

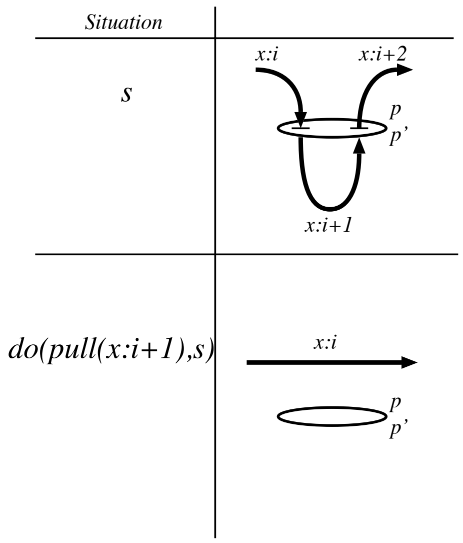


Figure 23: Removing the loop l.

More formally, let R be a hole and C_0 the state of a chain for a string S, where S passes through various holes, where each crossing is denoted by h_i^θ (for $i \in \mathbb{N}$ and $\theta \in \{+, -\}$):

$$C_0 = [h_1^i, h_2^j, \ldots, h_x^\alpha, h_y^\beta, \ldots, h_n^m].$$

Let's call w the string segment that falls in between the crossings h_x^α, h_y^β in the chain C_0, then $C_1 = do(pick(s : w, R^\epsilon), C_0)$ results in the creation of two opposite crossings (a loop) $\mathbf{R}^\epsilon$ and $\mathbf{R}^\lambda$ ($\epsilon \neq \lambda$) in the state C_1:

$$C_1 = [h_1^i, h_2^j, \ldots, h_x^\alpha, \mathbf{R}^\epsilon, \mathbf{R}^\lambda, h_y^\beta, \ldots, h_n^m].$$

The action *pull* is the reverse of *pick*, i.e. it removes a loop by passing all of it through the original hole. Thus, $C_2 = do(pull([R^\epsilon, R^\lambda], C_1)$ is equivalent to the original state:

$$C_2 = [h_1^i, h_2^j, \ldots, h_x^\alpha, h_y^\beta, \ldots, h_n^m].$$

If there were string crossings passing through the loop $l = [h_x^\alpha, h_y^\beta]$, then each of them, as an effect, would be *picked* towards the side of the whole where l was *pulled* towards.

An important side effect of pulling a loop is that adjacent loops will also disappear and this will make outer loops inherit their crossings[7]. As an example, consider the transition from Fig. 16(a) to Fig. 16(b) where we have pulled the loop formed by the pair of endpoints $[2, 3]$. As a result, loops $[1, 2]$ and $[3, 4]$ must also disappear because endpoints 2 and 3 will be removed. However, these two loops were inner loops of $[1, 4]$

[7]Appendix A contains a Prolog prototype that computes the effects of actions. In the case of action *pull*, predicate `replace_loops` takes account for this side effect.

which still remains after the pull action. This means that the dotted string crossing $[1,2]$ will be still crossing $[1,4]$ after pulling $[2,3]$.

The actions *pick* and *pull*, along with the action *pass* described in Section 2 above allow for the formal solution of the easy-does-it puzzle, as shown in the next section.

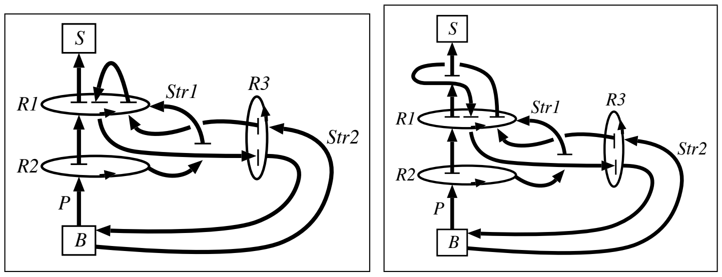
5 Working Example: Easy-does-it puzzle

This section presents a sequence of changes in the Easy-does-it puzzle's states that is one possible solution to the puzzle.

As described in Section 2, the initial state shown in Figure 9(b) corresponds to the chains in Formulae 1-3. We analyse next, step by step the puzzle solution. We refer to "steps" here (instead of actions) since sometimes, one physical step may imply several state transitions, as we will see next.

First step The first step of the solution (from state S_0 (Fig. 9(b)) to S_1 (Fig. 24(a))) involves picking the segment $Str2\!:\!1$ toward $R1^+$ on the state S_0, i.e., $do(pick(Str2:1, R1^+), S_0)$. This results in the state shown in Figure 24(a), with the following chain description.

$$\begin{aligned}
chain(P,S_1) &= [\underbrace{P_b^-}_{=B}, R2^+, R1^+, \underbrace{P_e^+}_{=S}];\\
chain(Str1,S_1) &= [\underbrace{Str1_b^-}_{=R2}, l(Str2, R3^+, [1,4])^+, \underbrace{Str1_e^+}_{=R1}];\\
chain(Str2,S_1) &= [\underbrace{Str2_b^-}_{=B}, \overbrace{R3^+, \underbrace{R1^+, R1^-}_{l(Str2,R1^+,[2,3])}, R3^-}^{l(Str_2,R_3+,[1,4])}, \underbrace{Str2_b^-}_{=B}].
\end{aligned}$$



(a) State S_1. (b) State S_2.

Figure 24: First and second transitions.

This process creates another loop in $Str2$ that can be seen as the pair $\langle R1^+, R1^-\rangle$ in $chain(Str2, S_1)$, and is denoted by $l(Str2, R1^+, [2,3])$. On the other hand, what was the loop $l(Str2, R3^+, [1,2])$ in S_0 has become $l(Str2, R3^+, [1,4])$ and is delimited now by $R3$, segment $Str2\!:\!1$, $R1$ and segment $Str2\!:\!3$.

Second step The second action is to pass the sphere S, and so the linked post tip P_e, towards $l(Str2, R1^+, [2,3])^+$, resulting in the state S_2 (Fig. 24(b)):

$$S_2 = do(pass(P_e^+, l(Str2, R1^+, [2,3])^+, S_1)$$

$$\begin{aligned}
chain(P, S_2) &= [P_b^-, R2^+, R1^+, l(Str2, R1^+, [2,3])^+, P_e^+]; \\
chain(Str1, S_2) &= [Str1_b^-, l(Str2, R3^+, [1,4])^+, Str1_e^+]; \\
chain(Str2, S_2) &= [Str2_b^-, R3^+, R1^+, R1^-, R3^-, Str2_b^+].
\end{aligned}$$

Third step Then, the loop $l(Str2, R1^+, [2,3])$ should be undone, by pulling it back towards $R1^-$ (downwards, in the diagram). In the physical puzzle, this movement would lead to the state S_3 depicted in Fig. 26(a). However, the step from S_2 to S_3 is not the result of a simple action on strings, since it also implies "sliding" $Str2$ down the post, which is not affected because it is a *rigid object*. Since, at our current representation level, we cannot represent rigidity, we must assume that the post is also a flexible string[8]. Under this assumption, the step from S_2 to S_3 is actually decomposed into the two transitions we describe next.

1. First, segment $Str2\!:\!2$ is pulled downwards $R1^-$, that is, formally:

   $$S_2' = do(pull(Str2\!:\!2, R1^-), S_2)$$

   As a result, we obtain Fig. 25 where the post is twisted forming two new loops: one below $R1$ (now crossed by $Str2$), and another single loop above $R1$. The chains we obtain in S_2' correspond to:

   $$\begin{aligned}
   chain(P, S_2') &= [P_b^-, R2^+, \overbrace{R1^+, R1^-}^{l(P,R1^+,[2,3])}\!\!\underbrace{, l(Str2, R3^+, [1,2])^+, R1^+}_{l(P,R1^-,[3,5])}, P_e^+]; \\
   chain(Str1, S_2') &= [Str1_b^-, l(Str2, R3^+, [1,2])^+, Str1_e^+]; \\
   chain(Str2, S_2') &= [Str2_b^-, R3^+, l(P, R1^-, [3,5])^+, R3^-, Str2_b^+].
   \end{aligned}$$

2. Second, we pull down the single loop in the post, $l(P, R1^+, [2,3])$, that was left alone above $R1$. Formally, this means pulling segment $P\!:\!2$ towards $R1^-$:

   $$S_3 = do(pull(P\!:\!2, R1^-), S_2)$$

   and we get the state S_3 in Fig. 26(a) described by the following chains:

   $$\begin{aligned}
   chain(P, S_3) &= [P_b^-, R2^+, l(Str2, R3^+, [1,2])^+, R1^+, P_e^+]; \\
   chain(Str1, S_3) &= [Str1_b^-, l(Str2, R3^+, [1,2])^+, Str1_e^+]; \\
   chain(Str2, S_3) &= [Str2_b^-, R3^+, R3^-, Str2_b^+].
   \end{aligned}$$

[8]In fact, the puzzle solution is not affected by using a string instead of a post. On the contrary, using a rigid post actually simplifies the puzzle, as it limits the possible actions on the strings, something that prunes the search space when physically exploring the puzzle.

An important observation is that destroying the upper loop $l(P, R1^+, [2,3])$ we had in the post at S_2' also destroys the adjacent loop $l(P, R1^-, [3,5])$ and, as a consequence, string $Str2$ is no longer crossing any loop at the resulting state S_3.

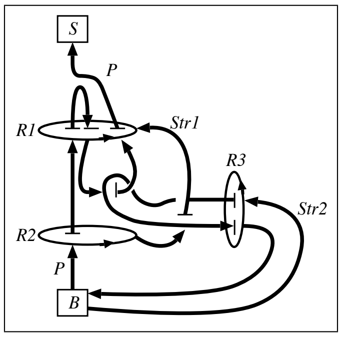
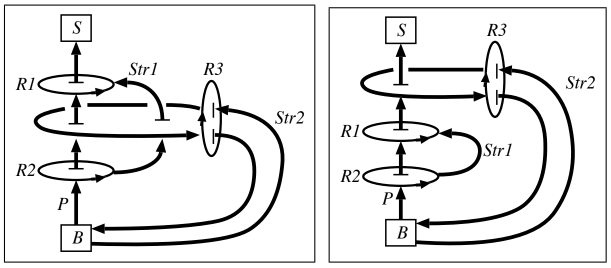


Figure 25: S_2': intermediate transition between S_2 and S_3 (assuming a flexible post).

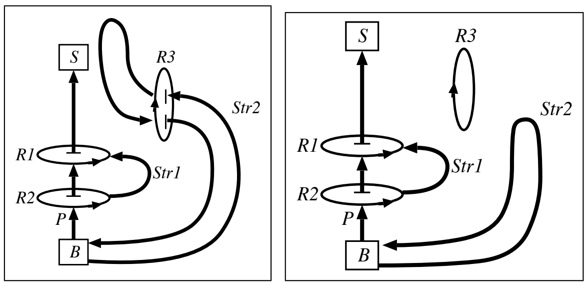


(a) State S_3. (b) State S_4.

Figure 26: Third and fourth transitions.

Fourth step This step is the combination of two movements: passing both the ring $R1$ and the tip $Str1_e^+$ (linked to the ring) downwards the loop in $Str2$. This is an action applied on a bundle of objects that must all be manipulated altogether [3]. Thus, we can perform the two actions $do(pass(R1, l(Str2, R3^+, [1,2])^-, S_3)$ and $do(pass(Str1_e^+, l(Str2, R3^+, [1,2])^-, S_3)$ in any order to get the resulting state S_4 depicted in Fig. 26(b) described by the chains:

$$\begin{aligned}
chain(P, S_4) &= [P_b^-, R2^+, R1^+, l(Str2, R3^+, [1,2])^+, P_e^+];\\
chain(Str1, S_4) &= [Str1_b^-, Str1_e^+];\\
chain(Str2, S_4) &= [Str2_b^-, R3^+, R3^-, Str2_b^+].
\end{aligned}$$



(a) State S_5. (b) State S_6.

Figure 27: Fifth and sixth transitions.

Fifth step The next action is to move the loop in $Str2$ upwards above the sphere, formally represented as $do(pass(P_e^+, l(Str2, R3^+, [1,2])^-), S_4)$, to obtain the state S_5 (Fig. 27(a)) with the chains:

$$\begin{aligned} chain(P, S_5) &= [P_b^-, R2^+, R1^+, P_e^+]; \\ chain(Str1, S_5) &= [Str1_b^-, Str1_e^+]; \\ chain(Str2, S_5) &= [Str2_b^-, R3^+, R3^-, Str2_b^+]. \end{aligned}$$

Sixth transition Finally, the loop in $Str2$, formed by $R3$, can be removed by a simple pull: $S_6 = do(pull(l(Str2, R3^+, [1,2]), R3^-), S_5)$, resulting in the following chain description, as represented in Figure 27(b):

$$\begin{aligned} chain(P, S_6) &= [P_b^-, R2^+, R1^+, P_e^+]; \\ chain(Str1, S_6) &= [Str1_b^-, Str1_e^+]; \\ chain(Str2, S_6) &= [Str2_b^-, Str2_b^+]. \end{aligned}$$

Note that the main ring $R3$ is not present in any of the last chain descriptions, so it is not crossed any more. Therefore, $R3$ is free from the set of entangled objects, solving the puzzle

6 A PROLOG simulator using the loop hierarchy

We have implemented a PROLOG prototype that allows simulation (temporal projection) for the strings domain in Section 4. The prototype is built to check the effects of the actions *pick* and *pull* on crossings and loops that can be formed or destroyed depending on the loop hierarchy. There are other features, however, that are not considered: all long objects are flexible and we do not consider constraints about which objects can pass through which holes. Besides, we do not handle the information about linked objects: this means that loops formed by linking two tips of a string to some object are not detected, but must be explicitly stated in the initial state. Similarly, multiple actions due to moving an object linked to another are not detected either and must be stated explicitly. We will include these features in a more complete tool still under development. Therefore, the prototype aims at verifying whether the formal definitions for the actions to create and destroy loops result in a coherent set of changes.

An important problem we have had to fix for building the prototype is related to reference numbers used in loops. In the example transitions of Section 5, the numbers A, B used inside loop references $l(S, H^x, [A, B])$ correspond to the *relative* positions in $chain(String, State)$ of the initial and final crossings forming the loop. We chose this representation because it is much simpler to check with respect to the diagrams. However, using relative positions has a serious drawback: since all actions create or destroy crossings in a chain, the relative position of a crossing may easily change. To put a simple example, note how the initial loop $l(Str2, R3^+, [1, 2])$ in the description of state S_0 of Easy-does-it (3) becomes $l(Str2, R3^+, [1, 4])$ at S_1 because two new crossings have been inserted in the middle, and the previous crossing number 2 in $chain(Str2, S_0)$ becomes now crossing number 4. This leads to a problem of elaboration tolerance since a part of the string not affected by some change must be explicitly renumbered. For instance, assume we have a string X with $n + 2$ crossings $C_1, \ldots, C_n, C_{n+1}, C_{n+2}$ where the two last crossings form a loop on ring R with $C_{n+1} = R^-$ and $C_{n+2} = R^+$. Assume also that a second string Y is crossing that loop:

$$\begin{array}{rcl} chain(X, S_0) & = & [S_b^-,\ C_1, \ldots, C_n, \underbrace{R^-, R^+}_{loop},\ S_e^+] \\ chain(Y, S_0) & = & [X_b^-,\ l(X, R^-, [n{+}1, n{+}2])^+,\ X_e^+] \end{array}$$

If we simply pass the tip S_b^- through a new hole H, the relative positions of all crossings will be increased in one and this implies renumbering all the potentially affected loops:

$$\begin{array}{rcl} chain(X, S_1) & = & [S_b^-,\ \underbrace{H^-}_{new}, C_1, \ldots, C_n, R^-, R^+,\ S_e^+] \\ chain(Y, S_1) & = & [X_b^-,\ l(X, R^-, [\underbrace{n{+}2, n{+}3}_{\text{renumbered}}])^+,\ X_e^+] \end{array}$$

Since a pull action may create and destroy a great number of loops, segments and crossings, it is practically impossible to keep in mind all the renumbering operations without potentially make a mistake in the program.

The solution to this problem is assigning a unique identifier from some set of labels Lb to each crossing in $chain(X, S)$ that does not vary when we insert or remove crossings from the chain. Since any segment can be subdivided as many times as needed, the set Lb should be an infinite, dense set. For convenience, we have taken the rational numbers $Lb = \mathbb{Q}$ for that purpose, since they additionally provide a complete linear ordering that can be used to represent the relative position of a segment with respect to the $chain(X, S)$. For each crossing C in $chain(X, S)$ we associate its identifier $i \in \mathbb{Q}$

with the notation[9] $C\!:\!i$. A simple solution to insert a crossing between two points A, B is taking the midpoint $(A+B)/2$ as new identifier. In this way, if we add the following crossing identifiers to our previous example:

$$\begin{aligned} chain(X, S_0) &= [S_b^- \!:\!0,\ C_1\!:\!1, \ldots, C_n\!:\!n, \underbrace{R^-\!:\!n{+}1, R^+\!:\!n{+}2}_{loop},\ S_e^+\!:\!n+3] \\ chain(Y, S_0) &= [X_b^-\!:\!0,\ l(X, R^-, [n{+}1, n{+}2])^+\!:\!1,\ X_e^+\!:\!2] \end{aligned}$$

then the result of passing X's tip just amounts to a minimum change in X while Y is *unaffected*:

$$\begin{aligned} &chain(X, S_1) = [S_b^-\!:\!0,\ \underbrace{H^-\!:\!1/2}_{\text{new crossing}}, C_1\!:\!1, \ldots, C_n\!:\!n, R^-\!:\!n{+}1, R^+\!:\!n{+}2,\ S_e^+\!:\!n{+}3] \\ &chain(Y, S_1) = chain(Y, S_0) \end{aligned}$$

Although absolute identifiers are internally used by the prototype, we have kept the use of relative values for the action descriptions. In this way, if we must specify a segment number to be pulled, or a loop to be crossed, we use the relative positions in the chain: we differentiate between $loop(S, A, B)$ to specify relative positions (used in the representation of actions) and $l(S, H^x, A, B)$ to specify absolute identifiers (internally used to compute the resulting state).

Appendix A shows the prototype source code. The execution of all steps in the Easy-does-it has been encoded as predicate `main` whose output is included below (for better readability, the second argument of chain/2 was edited by hand so that it represents the same states described in Section 5):

```
?- main.
chain(str1,s0)=[-s1b:0,+l(s2,+r3,1,2):1,+s1e:2]
chain(str2,s0)=[-s2b:0,+r3:1,-r3:2,+s2b:3]
chain(post,s0)=[-pb:0,+r2:1,+r1:2,+pe:3]
l(s2,-s2b,0,3)
l(s2,+r3,1,2)

Do pick(s2,1,+r1)

chain(str1,s1)=[-s1b:0,+l(s2,+r3,1,2):1,+s1e:2]
chain(str2,s1)=[-s2b:0,+r3:1,+r1:5 rdiv 4,
   -r1:7 rdiv 4,-r3:2,+s2b:3]
chain(post,s1)=[-pb:0,+r2:1,+r1:2,+pe:3]
l(s2,-s2b,0,3)
l(s2,+r3,1,2)
l(s2,+r1,5 rdiv 4,7 rdiv 4)

Do pass_tip(post,end,+loop(s2,2,3))

chain(str1,s2)=[-s1b:0,+l(s2,+r3,1,2):1,+s1e:2]
chain(str2,s2)=[-s2b:0,+r3:1,+r1:5 rdiv 4,
   -r1:7 rdiv 4,-r3:2,+s2b:3]
chain(post,s2)=[-pb:0,+r2:1,+r1:2,
```

[9]Although it is the same notation used for segments, note that there is no ambiguity, since in $C\!:\!i$ the left argument is a crossing rather than a string.

```
  +l(s2,+r1,5 rdiv 4,7 rdiv 4):5 rdiv 2,+pe:3]
l(s2,-s2b,0,3)
l(s2,+r3,1,2)
l(s2,+r1,5 rdiv 4,7 rdiv 4)

Do pull(s2,2)

chain(str1,s2')=[-s1b:0,+l(s2,+r3,1,2):1,+s1e:2]
chain(str2,s2')=[-s2b:0,+r3:1,
   -l(post,-r1,9 rdiv 4,11 rdiv 4):3 rdiv 2,-r3:2,+s2b:3]
chain(post,s2')=[-pb:0,+r2:1,+r1:2,-r1:9 rdiv 4,
   +l(s2,+r3,1,2):5 rdiv 2,+r1:11 rdiv 4,+pe:3]
l(s2,-s2b,0,3)
l(s2,+r3,1,2)
l(post,+r1,2,9 rdiv 4)
l(post,-r1,9 rdiv 4,11 rdiv 4)

Do pull(post,2)

chain(str1,s3)=[-s1b:0,+l(s2,+r3,1,2):1,+s1e:2]
chain(str2,s3)=[-s2b:0,+r3:1,-r3:2,+s2b:3]
chain(post,s3)=[-pb:0,+r2:1,+l(s2,+r3,1,2):5 rdiv 2,
   +r1:11 rdiv 4,+pe:3]
l(s2,-s2b,0,3)
l(s2,+r3,1,2)

Do pass_tip(s1,end,-loop(s2,1,2))

chain(str1,s4')=[-s1b:0,+s1e:2]
chain(str2,s4')=[-s2b:0,+r3:1,-r3:2,+s2b:3]
chain(post,s4')=[-pb:0,+r2:1,+l(s2,+r3,1,2):5 rdiv 2,
   +r1:11 rdiv 4,+pe:3]
l(s2,-s2b,0,3)
l(s2,+r3,1,2)

Do pass_hole(r1,-loop(s2,1,2))

chain(str1,s4)=[-s1b:0,+s1e:2]
chain(str2,s4)=[-s2b:0,+r3:1,-r3:2,+s2b:3]
chain(post,s4)=[-pb:0,+r2:1,+r1:5 rdiv 2,
   +l(s2,+r3,1,2):11 rdiv 4,+pe:3]
l(s2,-s2b,0,3)
l(s2,+r3,1,2)

Do pass_tip(post,end,-loop(s2,1,2))

chain(str1,s5)=[-s1b:0,+s1e:2]
chain(str2,s5)=[-s2b:0,+r3:1,-r3:2,+s2b:3]
chain(post,s5)=[-pb:0,+r2:1,+r1:5 rdiv 2,+pe:3]
l(s2,-s2b,0,3)
l(s2,+r3,1,2)

Do pull(s2,1)

chain(str1,s6)=[-s1b:0,+s1e:2]
chain(str2,s6)=[-s2b:0,+s2b:3]
chain(post,s6)=[-pb:0,+r2:1,+r1:5 rdiv 2,+pe:3]
```

```
l(s2,-s2b,0,3)

true
```

7 A Model for Chains within Knot Theory

In this section we justify the representation described above by proving that for any chain description there is a translation in terms of a sequence of string crossings (i.e., intersections of lines in the diagrams representing the domain). We also show that for any action executed on the chain representation there is a corresponding sequence of movements on the string intersections that results in the same state. In order to do that, we need to introduce a number of definitions from the subfield of topology that studies mathematical knots: Knot Theory [11].

In the remainder of this paper we use the term *string intersection* (or only *intersection*) to refer to string crossings, in order to differentiate it from hole crossings.

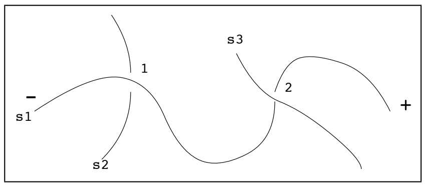
This section introduces a notation for string intersections based on the work presented in [28], that proposes a representation of the various shapes a string assumes by collecting the points where the string crosses itself, scanning it from one of its terminals to the other. This representation, called p-data, is based on a 2D projection (knot diagram [11]) of a 3D knot. Inspired by the p-data representation, this section introduces a representation for string intersections that takes into account cases where a string intersects another string, besides itself.

The string-intersection representation used in this work is constructed as follows. The direction in which the string is swept needs to be defined, in this work it goes from its beginning to its ending tips. From one terminal of the string to the other, each point where the string crosses another string (or itself) is included in a list. At each of these points the direction of the intersection is annotated (represented by the *sign* of a cross product, as defined below), the other string involved in the intersection is also annotated, along with information about the *relative vertical position* of an intersection. The vertical position of the intersection is determined by verifying the position of each string at each intersection found, following the direction in which the string is being traversed. We say that an intersection is *upper* (U) if at the intersection point the string that is being traversed is on top; analogously, a *lower* (L) intersection is that where at the intersection point the string that is being traversed is below the other.

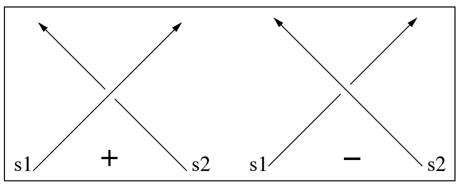
Figure 28(a) shows examples of upper and lower intersections related to a string $s1$ with respect to strings $s2$ and $s3$. The intersection marked with 1 is an upper intersection of $s1$ with respect to $s2$, and is denoted as U_{s2} (if $s1$ is traversed) and the intersection marked with 2 is a lower intersection of $s1$ with respect to $s3$, and is denoted as L_{s3}.

The direction of the intersection (or its *sign*) can be obtained from the sign of the cross product between the direction of the string that is on top (the upper string) and the direction of the string that is below (the lower string) at the intersection. More formally: $(\vec{l}_{upper} \times \vec{l}_{lower})$, where $\vec{l}_{upper}$ and $\vec{l}_{lower}$ are, respectively, the directions of the upper and lower strings at the intersection point (cf. Figure 28(b)).

Therefore, an elementary loop created by two hole crossings of a string St through a



(a)



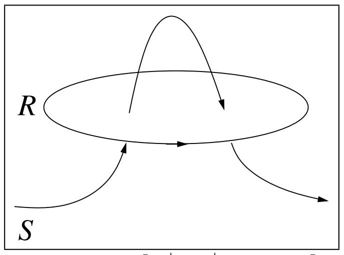
(b)

Figure 28: String-intersection representation.

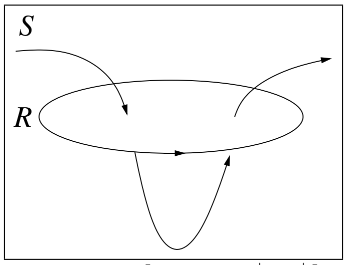
ring R (shown in Figure 29(a)), whose chain description is $chain(St, S) = [R^+, R^-]$, has the following string-intersection (SI) description:

$$SI(St) = [L_R^+, U_R^+, U_R^-, L_R^-].$$

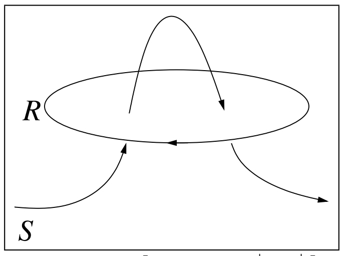
In terms of string intersections, the way the diagram is drawn changes the description. There are only four ways to draw an elementary loop through a ring, shown in Figure 29 (other depictions are achieved by rotating the ones shown in the picture). Note that a hole crossing R^+ is written as the pair $\langle L_R^+, U_R^+\rangle$, for rings oriented in anti-clockwise direction, and $\langle U_R^+, L_R^+\rangle$, for rings oriented in clockwise direction (analogously for R^-).



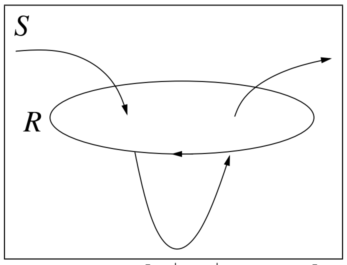
(a) $SI(S) = [L_R^+, U_R^+, U_R^-, L_R^-]$



(b) $SI(S) = [U_R^-, L_R^-, L_R^+, U_R^+]$



(c) $SI(S) = [L_R^-, U_R^-, U_R^+, L_R^+]$



(d) $SI(S) = [U_R^+, L_R^+, L_R^-, U_R^-]$

Figure 29: Distinct ways of drawing elementary loops and their related string-crossing representations.

In this work we use the three basic actions on strings introduced in Knot Theory, called Reidemeister moves [19, 11]. Figure 30 shows the three Reidemeister moves for non-oriented strings: type I (Fig. 30(a)), type II (Fig. 30(b)) and type III (Fig. 30(c)). These actions can be described as follows: type I adds or deletes a simple twist in the string; Type II allows the inclusion (or exclusion) of two crossings in the string; Type III slides a strand of the string from one side of a crossing to the other. Reidemeister moves types II and III for oriented strings are respectively shown in Figures 31 and 32 [30] (type I is not shown since it is not used in this work).

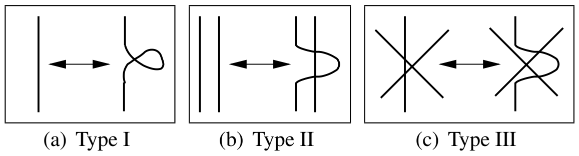
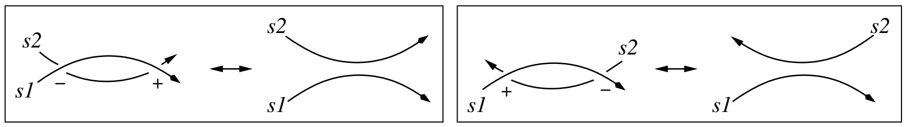

(a) Type I (b) Type II (c) Type III

Figure 30: The Reidemeister moves.



(a) Type IIa (b) Type IIb

Figure 31: Reidemeister move type II.

Reidemeister move type II (Fig. 31), for a string $s1$ can be represented in terms of string intersections as:

$$[X^i_\alpha, X^j_\alpha] \leftrightarrow [\ ],$$

where $X \in \{L, U\}$, α is a string (or a section of a single string), and i and j ($i \neq j$) are the signs of the crossings. E.g. the string intersection representation for the case shown in Figure 31(a) can be written as $[U^-_{S2}, U^+_{S2}] \leftrightarrow [\ ]$

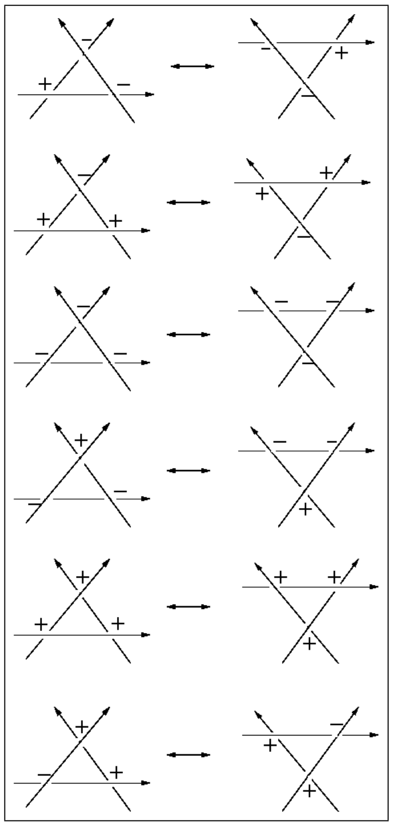
Similarly, Reidemeister move type III, for oriented strings (shown in Figure 32), can be represented in terms of string intersections as:

$$[X^i_\alpha, Y^j_\beta] \leftrightarrow [Y^j_\beta, X^i_\alpha],$$

where $X, Y \in \{L, U\}$, α and β are the strings (or distinct sections of a single string) involved in the crossings, and i and j are the signs of the crossings, as exemplified in Figure 34 for three strings $s1$, $s2$ and $s3$.

Besides Reidemeister moves, as our domain has open-ended strings, we have to consider also the *cross* move introduced in [28], that adds or removes an intersection by *passing* the open end of a string on top of another string (or on a section of the same string). Figure 33 represents the cross move.

With the formalism described above, it is now possible to show that for any *pass*, *pick* and *pull* action on the original chain description, there is an equivalent finite sequence of Reidemeister moves on the related string-intersection translation. The concept of equivalence here means that the states resulting from *pass*, *pick* and *pull*, when translated to string intersections, are the same as the ones achieved by a sequence of


(a) Type IIIa

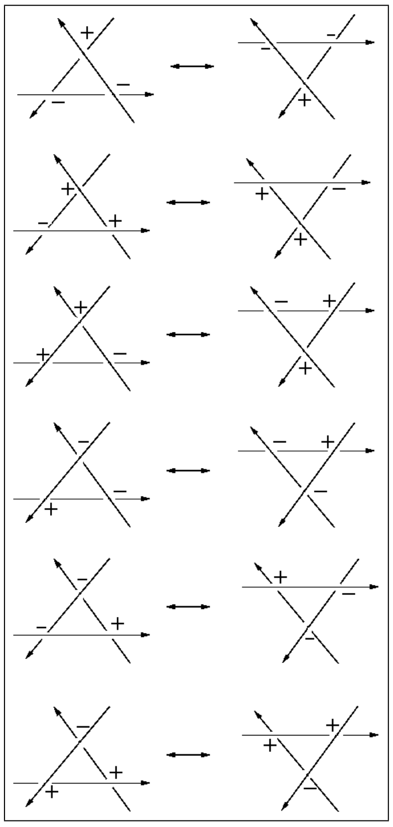

(b) Type IIIb

Figure 32: Reidemeister move type III (adapted from [30]).

Reidemeister moves. Thus, we say that the chain representation (and its related actions) is *sound* with respect to Reidemeister moves in knot theory.

In order to prove soundness of the hole crossing representation, and its related actions *pass*, *pick* and *pull*, with respect to string intersections and Reidemeister moves we need to prove first that every loop in a loop hierarchy can be reduced to an *elementary loop*, i.e. a loop that does not have any related inner loops (Lemma 1); and, after that, we need to prove that the translation from the chain representation to the string intersection representation is unique (Lemma 2). Soundness is proved in Theorem 1.

Lemma 1. *Every loop, that is not a ring, can be reduced to an elementary loop by a sequence of* pull *actions.*

Proof. This follows by induction on the number of inner loops related to an outer loop, observing that (in this work) loops are assumed to be only created by *pick* actions. □

Lemma 2 (Uniqueness of translation)**.** *Any hole crossing in a chain description can be translated into one and only one sequence of string crossings.*

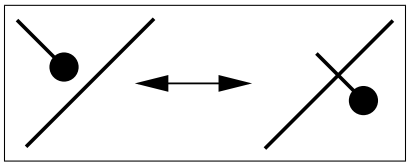
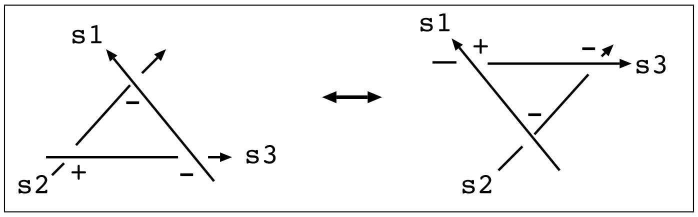


Figure 33: Cross move [28].



$$
\begin{array}{lcl}
SI(s1) = [U^-_{s3}, U^-_{s2}] & \leftrightarrow & SI(s1) = [U^-_{s2}, U^-_{s3}] \\
SI(s2) = [L^+_{s3}, L^-_{s1}] & \leftrightarrow & SI(s2) = [L^-_{s1}, L^+_{s3}] \\
SI(s3) = [U^+_{s2}, L^-_{s1}] & \leftrightarrow & SI(s3) = [L^-_{s1}, U^+_{s2}]
\end{array}
$$

Figure 34: Reidemeister move type III

Proof. For the purpose of *reductio ad absurdum*, let's assume that there is a representation for an elementary loop that is not one of the four shown in Figure 29. In this case, either the indexes, the superscripts or the crossings would be distinct from the ones shown in the figure. In each of these cases the representation is not a loop. □

Theorem 1 (Soundness). *Let c be any chain and p_c be a translation of c in terms of string intersections. Let also a be a* pass*,* pick *or* pull *action on c that changes c into a successor c_1 (where p_{c_1} is the translation of c_1 in terms of string intersections). Then, there is a finite sequence of Reidemeister moves that, when applied on p_c, results in a state p'_c such that $p'_c = p_{c1}$.*

Proof. We first show that any elementary, *pass*, *pull* and *pick* action can be described by a sequence of Reidemeister moves on the related string-crossing representation. From lemma 1 this result can be applied to any *pull* and *pick* actions.

The pass action can be directly translated by a sequence of two cross moves. Similarly, the pick action, and the most basic form of *pull* (in cases where there is no string crossing the loops involved in this action), can be described by the Reidemeister move type II (applied twice, one on the front and another on the back sides of the hole), as shown in Figure 35.

We now consider the cases where there are strings crossing the loops involved in the pull action. We show the sequence of Reidemeister moves for a $pull$ action for the case where there is only one string crossing a loop. The case of n ($n \in \mathbb{N}$) strings crossing a loop follows by induction on n.

Consider the loop represented in Figure 36(a), that is formed by string $S1$ and ring

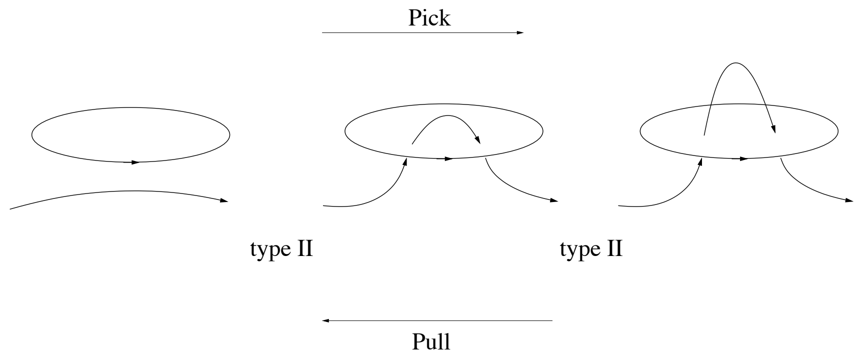


Figure 35: The basic cases of pick and pull can be described as Reidemeister type II moves.

R (for brevity, only one side of the ring is shown in this figure, the derivation for the other side is analogous). This loop is crossed by another string $S2$. The scenario shown in Figure 36(a) has the following chain descriptions:

$$chain(S1) = [R^+, R^-], \qquad chain(S2) = [l(S1, R^+, [C, D])^+]. \tag{9}$$

A pull action on the loop $l(S1, R^+, [C, D])$ on this state deletes the loop and applies a pick action on the strings that where crossing $l(S1, R^+, [C, D])$ (in this case on the string $S2$), leading to the state shown in Figure 36(c), that has the following chain description:

$$chain(S1) = [l(S2, R^-, [G, H])^-], \qquad chain(S2) = [R^-, R^+]. \tag{10}$$

Translating Formulae 9 to the string-crossing notation, we obtain the Formulae 11 below.

$$SI(S1) = [L^+_R, L^+_{S2}, U^+_{S2}, L^-_R], \qquad SI(S2) = [U^+_{S1}, L^+_{S1}]. \tag{11}$$

The task now is to find a finite sequence of Reidemeister moves on Formulae 11 that leads to a state description equivalent to that represented by Formulae 10. By applying Reidemeister move type III on the string-crossings A and B, and Reidemeister move type II on the string $S2$ (at the point of the crossings A and B), the state shown in Figure 36(b) is achieved, whose description is shown in Formulae 12. Reidemeister move type II introduced two new loops on string $S2$, representing the *pick* action related to *pulling* a loop where there is a crossing.

$$SI(S1) = [L^+_{S2}, L^+_R, L^-_R, U^+_{S2}]. \quad SI(S2) = [L^-_R, U^+_{S1}, L^+_R, L^-_R, L^+_{S1}, L^+_R]. \tag{12}$$

Then, Reidemeister move II is applied on the state shown in Figure 36(b), first deleting the loop $[E, F]$ (on string $S2$, defined by the pair $\langle L^+_R, L^-_R \rangle$: 1^{st} and 3^{rd}

elements of the list $SI(S2)$ in Formulae 12), and then deleting the loop $[C, D]$ (on string $S1$, defined by the pair $\langle L^-_R, L^+_R \rangle$: 4^{th} and 6^{th} elements of the list $SI(S1)$ in Formulae 12). This results in the state shown in Figure 36(c), whose string-intersection description is represented in Formulae 13 below.

$$SI(S1) = [L^-_{S2}, U^-_{S2}] \qquad\qquad SI(S2) = [L^-_R, U^-_{S1}, L^-_{S1}, L^+_R] \tag{13}$$

which is equivalent to the direct description of Formulae 10 in terms of string intersections, excluding the intersections that are not related to holes.

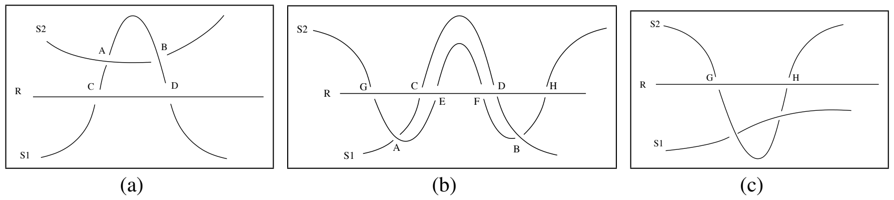


Figure 36: pull.

As the translation is unique (cf. Lemma 2), this shows that actions on the chain representation are sound with respect to Reidemeister moves on Knot Theory.

□

8 Discussion and Conclusion

This paper is part of our long-term project whose aim is to formalise, and provide automated solutions, for spatial puzzles of increasing complexity [2, 20, 21, 22, 3, 23, 25, 24]. In the present work we considered a puzzle in which the manipulation of loops formed by strings is an essential part of the solution, an issue that was open in our previous investigations [25].

Loops on strings are defined in this work as pairs of crossings made by a single string on the same hole but in distinct directions. The consideration of loops brought to light two interesting points related to the ontology of objects. First of all, string loops can be used as holes, conversely (but not explored in this paper) a hole made from a flexible object, if it is cut at one point, behaves as a string. The similar use of holes and string loops was formalised by introducing a function l to represent the loop as a hole, when it is used as such. Therefore, the function l connects the basic definition of loops (as pairs of crossings on a chain) to our previous representation of hole and hole crossings. Second, when a loop, that is part of a set of loops, is crossed by a string (or any longer object) the actions applied to this loop may have side effects on the objects that are currently crossing it. This issue was solved in the present paper by assuming a hierarchy of loops, in which the crossings related to a loop are inherited by other loops in the hierarchy. Both, the analogous use of holes and loops and the loop hierarchy are key aspects on reasoning about flexible objects that (to the best of our knowledge) were first discussed in this paper.

Another contribution of this paper was the translation from the chain representation of string puzzles to a representation based on string crossings (or intersections). This enabled us to prove that the solutions provided by manipulating a puzzle domain represented as chains have a model in the more general framework of knot theory. Although this does not show that the solutions are sound with respect to the physical manipulations of the objects involved in the domain, this proves that our solution is correct with respect to the diagrammatic representation of knots in knot theory. Proving soundness with respect to the physical world turns out to be a very elusive task [16, 8].

The proof of soundness presented in this paper is based on showing that for every action on a chain, there is a translation in terms of Reidemeister moves. Conversely, a proof of *completeness* in this domain would have to show that for every sequence of Reidemeister moves on a puzzle description, there is a translation in terms of actions on the equivalent chain. This, however, cannot be proved since Reidemeister moves are applied on string intersections and, thus, have a greater freedom to operate changes on the puzzle domain than actions on chains (that are restricted to passing objects through holes, or creating/destroying loops).

The formal solution given in this paper was also checked by means of an implemented prototype of a simulator that was used to verify that the actions formalised had the intended effects on the domain considered in this work. The search for an algorithm for actually solving puzzles automatically falls within the AI planning field of research. In [3] we proposed a simple iterative deepening blind search planner that was capable of finding solutions to the Fisherman's Folly puzzle automatically, but timed out for more complex instances of similar puzzles. As we discussed in [24], the design of an efficient planner for this kind of problems is a challenging research topic since the domain is potentially unbounded (new loops and crossings can be created all the time). This is not usual in standard planning and will probably make it more difficult to find a good heuristic function to prune the search space.

Although knot theory was an important part of this paper, we left the possibility of tying knots in our domain for future research. Although essential for solving most real world problems (such as tying a shoelace, operating a sailboat or executing sutures), the assumption of knots greatly increases the complexity of automated problem solving with strings. We believe that the solution resides on a proper use of heuristics to drive the process of knot tying, but a way of integrating heuristics in the domains considered in this paper is still under investigation.

Future work will also consider the computational complexity of our approach. We believe that this is related to the complexity of the *unknotting problem* in knot theory, which falls within the class of NP problems [10]. Another related question for further investigation is whether there is a formal account of a complexity hierarchy for the classes of puzzles this paper considered. We have an informal classification of them in terms of *the kinds of actions* required to solve the different puzzles: for instance, some of the puzzles do not require loops (Fisherman's Folly and others), others require loops but not knots (Easy-does-it and others), and other puzzles do require some knot manipulation (which is our immediate next step). But a formal classification, or a complexity assessment, of such spatial puzzles are still open issues.

Acknowledgements: We are specially thankful to the reviewers since they have definitely helped to improve the paper with their interesting suggestions and their careful proof reading. Paulo E. Santos acknowledges financial support from FAPESP grant 2012/04089-3, and CNPq “bolsa de produtividade em pesquisa” grant 307093/2014-0. Pedro Cabalar was partially supported by Spanish MEC project TIN2013-42149-P. This work was developed during the visit of Paulo Santos to Coruña financed by the Santander Universia Young Investigator’s Award.

References

[1] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[2] P. Cabalar and P. Santos. Strings and holes: an exercise on spatial reasoning. In *Proc. of the 10th Ibero-American Artificial Intelligence Conference (IBERAMIA’06)*, volume 4140 of *Lecture Notes in Artificial Intelligence*, pages 419–429. Springer, Ribeirão Preto, Brazil, October 2006.

[3] Pedro Cabalar and Paulo E. Santos. Formalising the fisherman’s folly puzzle. *Artificial Intelligence*, 175(1):346–377, January 2011.

[4] R. Casati. Knowledge of knots: shapes in action. In *Shapes 2.0: The Shapes of Things*, volume 1007 of *CEUR Workshop Proceedings*, 2013. http://ceur-ws.org/Vol-1007.

[5] R. Casati and A. C. Varzi. *Holes and Other Superficialities*. MIT Press, Cambridge, MA, 1994.

[6] R. Casati and A. C. Varzi. *Parts and Places*. MIT press, 1999.

[7] A. G. Cohn and J. Renz. Qualitative spatial representation and reasoning. In F. van Hermelen, V. Lifschitz, and B. Porter, editors, *Handbook of Knowledge Representation*, pages 551–596. Elsevier, 2008.

[8] E. Davis. An ontology of physical actions. Technical Report 123, Computer Science Dept., New York Univ., 1984.

[9] M. Gelfond and V. Lifschitz. The stable models semantics for logic programming. In *Proc. of the 5th Intl. Conf. on Logic Programming*, pages 1070–1080, 1988.

[10] Joel Hass, Jeffrey C. Lagarias, and Nicholas Pippenger. The computational complexity of knot and link problems. *J. ACM*, 46(2):185–211, March 1999.

[11] L. H. Kauffman. *Formal Knot Theory*. Princeton University Press, 1983.

[12] Roman Kontchakov, Agi Kurucz, Frank Wolter, and Michael Zakharyaschev. Spatial logic + temporal logic = ? In Marco Aiello, Ian Pratt-Hartmann, and Johan Van Benthem, editors, *Handbook of Spatial Logics*, pages 497–564. Springer Netherlands, 2007.

[13] S.C. Levine, K.R. Ratliff, J. Huttenlocher, and J. Cannon. Early puzzle play: a predictor of preschoolers' spatial transformation skill. *Developmental Psychology*, 48(2):530–42, 2012.

[14] Gérard Ligozat. *Qualitative Spatial and Temporal Reasoning*. John Wiley & Sons, 2013.

[15] J. McCarthy. AI as sport. *Science*, 276(5318):1518 – 1519, 1997.

[16] J. McCarthy. Approximate objects and approximate theories. In *KR2000: Principles of Knowledge Representation and Reasoning,Proceedings of the Seventh International conference*, pages 519–26. Morgan-Kaufman, 2000.

[17] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence Journal*, 4:463–512, 1969.

[18] David Pearce. A new logical characterisation of stable models and answer sets. In *Non monotonic extensions of logic programming. Proc. NMELP'96. (LNAI 1216)*. Springer-Verlag, 1996.

[19] K. Reidemeister. *Knot Theory*. BCS Associates, 1983.

[20] Paulo E. Santos and Pedro Cabalar. Passing through holes and getting entangled by strings: An automated solution for a spatial puzzle. In *Proc. of the Workshop on Spatial and Temporal Reasoning, (inside the European Conference on Artificial Intelligence, ECAI'06)*, Riva del Garda, Italy, August 2006.

[21] Paulo E. Santos and Pedro Cabalar. Holes, knots and shapes: A spatial ontology of a puzzle. In *Proc. of the 8th International Symposium on Logical Formalizations of Commonsense Reasoning (Commonsense'07), AAAI Spring Symposium Series 2007*, Stanford, CA, USA, March 2007.

[22] Paulo E. Santos and Pedro Cabalar. The space within fisherman's folly: Playing with a puzzle in mereotopology. *Spatial Cognition & Computation*, 8(1-2):47–64, 2008.

[23] Paulo E. Santos and Pedro Cabalar. Knots world: an investigation of actions, change and space. In *Proc. of the Spatio-temporal Dynamics Workshop (STeDy'12) (inside ECAI'12)*, pages 36–44, Montpellier, France, 2012.

[24] Paulo E. Santos and Pedro Cabalar. An investigation of actions, change, space. In *Proc. of the 23rd International Conference on Automated Planning and Scheduling (ICAPS 2013)*, Rome, Italy, June 2013.

[25] Paulo E. Santos and Pedro Cabalar. An investigation of actions, change, space within a hole-loop dichotomy. In *Proc. of the 11th Intl. Symp. on Logical Formalizations of Commonsense Reasoning (Commonsense'13)*, Ayia Napa, Cyprus, May 2013.

[26] H. Simon and J. Schaeffer. The game of chess. In Hart Aumann, editor, *Handbook of Game Theory with Economic applications*, volume 1, pages 1–17. North Holland, 1992.

[27] O. Stock, editor. *Spatial and Temporal Reasoning*. Kluwer Academic, 1997.

[28] J. Takamatsu, T. Morita, K. Ogawara, H. Kimura, and K. Ikeuchi. Representation for knot-tying tasks. *IEEE Transactions on Robotics and Automation*, 22(1):65 – 78, 2006.

[29] Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors. *Handbook of Knowledge Representation (Foundations of Artificial Intelligence)*. Elsevier Science, 2007.

[30] F. Wu. Knot theory and statistical mechanics. *Rev. Mod. Phys.*, 64:1099–1131, Oct 1992.

Appendix A. Prolog prototype

```
:- dynamic newxing/1.

main :-
    initial(easydoesit,T),
    complete_state(T,T0),
    write_state(T0),nl,
    execute(T0,[
      pick(s2,1,+r1),
      pass_tip(post,end,+loop(s2,2,3)),
      pull(s2,2),
      pull(post,2),
      pass_tip(s1,end,-loop(s2,1,2)),
      pass_hole(r1,-loop(s2,1,2)),
      pass_tip(post,end,-loop(s2,1,2)),
      pull(s2,1)
    ],_T1),
    true.

initial(easydoesit,[
  chain(s1)=[-s1b,+l(s2,+r3,1,2),+s1e],
  chain(s2)=[-s2b,+r3,-r3,+s2b],
  chain(post)=[-pb,+r2,+r1,+pe]
]).

complete_state(T,TN):-
    change_fluents(all_add_xing_ids,_,T,T1),
    update_loops(T1,TN).

% change_fluents(Pred,Data,T,TN)
% Changes all fluent values in state T using predicate Pred with data Data
% to produce new state T1. If Pred fails, fluent values are left unchanged
change_fluents(_,_,[],[]):-!.
change_fluents(Pred,Data,[F=V|T],[F=V1|TN]) :-
    Call =.. [Pred,F=V,Data,V1], Call,!, change_fluents(Pred,Data,T,TN).
change_fluents(Pred,Data,[Fact|T],[Fact|TN]) :- change_fluents(Pred,Data,T,TN).
```

```
% Apply add_xing_ids on all strings
all_add_xing_ids(chain(_)=Zs,_,Ws) :- !, add_xing_ids(Zs,Ws).

% Generate a xing identifier (its position number) for each crossing
add_xing_ids(Xs,Ys) :- add_xing_ids(0,Xs,Ys).
add_xing_ids(_,[],[]) :- !.
add_xing_ids(N,[X|Xs],[X:N|Ys]) :- N1 is N+1, add_xing_ids(N1,Xs,Ys).

% Add the current list of loops to the state T, yielding TN
update_loops(T,TN):-
    findall(Fact,(member(Fact,T), \+ (Fact = l(_,_,_,_))), T1),
    findall(l(S,X,A,B),(member(chain(S)=Zs,T1), get_loop(Zs,l(X,A,B))),T2),
    append(T1,T2,TN).

% get_loop(Zs,L)
% It gets each possible loop L from an initial list of (identified)
% crossings Zs. Each loop is like l(X,A,B) where X:A is the initial
% crossing and Y:B is the (opposite) final crossing
% ?- ex(1,Xs), get_loop(Xs,L).

get_loop([X:A|Zs],l(X,A,B)) :- find_opposite(X,Zs,B).
get_loop([_|Zs],L) :- get_loop(Zs,L).

find_opposite(X,[Y:B|_],B) :- opposite(X,Y).
find_opposite(X,[_|Zs],B) :- find_opposite(X,Zs,B).

% Executes a list of actions As on an initial state T0 to reach state TN
execute(T0,[],T0) :- !.
execute(T0,[A|As],TN) :-
    writeall(['Do ',A,'\n\n']),
    do(T0,A,T1),
    write_state(T1),nl,
    execute(T1,As,TN).

% do(T0,A,TN) - Execute a single action A on T0 to yield TN

do(T0,pick(S,N,X),TN) :-
    change_fluents(do_pick,pick(S,N,X),T0,T1),
    update_loops(T1,TN).

do(T0,pull(S,N),TN) :- do_pull(S,N,T0,TN).

% If we pass the tip through a loop, we just use the loop crossing
% positions, not their ids

do(T0,pass_tip(S,T,X),TN) :-
    get_loop_id(T0,X,W),!,
    do(T0,pass_tip(S,T,W),TN).

do(T0,pass_tip(S,T,X),TN) :-
    change_fluents(do_pass_tip,(S,T,X),T0,TN).

do(T0,pass_hole(H,P),TN) :-
    get_loop_id(T0,P,W),!,
    change_fluents(do_pass_hole,pass_hole(H,W),T0,T1),
    update_loops(T1,TN).
```

```
do(T0,skip,T0) :- !.

% Pass tip T of string S with chain Zs towards X. T can be "begin" or "end"
do_pass_tip(chain(S)=Zs,(S,end,X),ZsN) :-
    append(Zs0,[(Y:A),(W:B)],Zs),                % Take last crossing but
                                                 % one (Y:A). (W:B) is the tip
  ( opposite(X,Y),!,append(Zs0,[(W:B)],ZsN)      % (PR) Remove last crossing (Y:A)
   ; midpoint(A,B,M),append(Zs0,[(Y:A),(X:M),(W:B)],ZsN) ).
                                                 % (PL) insert a new crossing

% NOTE: for simplicity, passing tips to the opposite side of a hole
% never creates new loops. We pass the tip "pulling" the string with it
do_pass_tip(chain(S)=[(W:A),(X:_)|Zs],(S,begin,X),[(W:A)|Zs]) :- !.    % (NL)
do_pass_tip(chain(S)=[(W:A),(Y:B)|Zs],(S,begin,X),[(W:A),(X1:M),(Y:B)|Zs]) :-
    opposite(X,X1),midpoint(A,B,M).                                     % (NR)

% passhole(H,P,Zs,ZsN)
% passholed object H to face P changes crossings Zs to ZsN

passhole(_,_, [Z],[Z]) :- !.
passhole(H,P, [Z,(X:A)|Zs],[Z|ZsN]) :-
    xing_sign_hole(X,_,G), G\=H, !, passhole(H,P, [(X:A)|Zs],ZsN).

passhole(H,P, [(X:A),(W:B),(Y:D)|Zs],[(X:A),(P:M),(W:B),(P1:M1)|ZsN]) :-
    xing_sign_hole(X,_,H),opposite(P,P1),X \= P1, Y \= P, !,
    midpoint(A,B,M), midpoint(B,D,M1),
    passhole(H,P, [(Y:D)|Zs],ZsN).       % (1R)

passhole(H,P, [(P1:_),(X:B),(P:C)|Zs], [(X:B)|ZsN]) :-
    xing_sign_hole(X,_,H),opposite(P,P1),!,
    passhole(H,P, [(P:C)|Zs],L), L=[(P:C)|ZsN].             % (1L)

passhole(H,P, [(X:A),(W:B),(P:D)|Zs],[(X:A),(P:B),(W:D)|ZsN]) :-
    xing_sign_hole(W,_,H),opposite(P,P1), X \= P1, !,
    passhole(H,P, [(P:D)|Zs],L), L=[(P:D)|ZsN].  % (2R)

passhole(H,P, [(P1:A),(W:B),(Y:D)|Zs], [(W:A),(P1:B)|ZsN]) :-
    xing_sign_hole(W,_,H),opposite(P,P1),Y \= P,!,
    passhole(H,P, [(Y:D)|Zs],ZsN).                 % (2L)

% To be used with change_fluents
do_pass_hole(chain(_)=Zs,pass_hole(H,P),Ws) :-
    passhole(H,P,Zs,Ws).


% Execution of a pick on string S with chain of crossings Zs after crossing
% number N (beginning with 0) towards X and backwards and generates new crossings
% Ws and a new loop l(S,X,C,D). We assume that N is at least 0 and at most
% length(Zs)-2 (i.e. string tips are included)
% MidPT is a midpoint to place in the middle of the new crossings (left and right).
% If MidPT=mid then the midpoint is (A+B)/2

pick(S,Zs,N,X,l(S,X,C,D),MidPT,Ws) :-
    opposite(X,Y),
    append(Zs0,[(Z1:A),(Z2:B) | Zs1],Zs), length(Zs0,N),  % split Zs
    (MidPT = mid,!,midpoint(A,B,M); M=MidPT),
    midpoint(A,M,C),midpoint(M,B,D),
    append(Zs0,[(Z1:A),(X:C),(Y:D),(Z2:B)|Zs1],Ws).

midpoint(A,B,M) :- M is (A+B) rdiv 2.

% Version for using with "change_fluents". We add the new loop as a fact
% for "newloop" dynamic predicate
do_pick(chain(S)=Zs,pick(S,N,X),Ws) :-
    pick(S,Zs,N,X,_,mid,Ws).

% Pulling a "clean" loop on crossing number N (begins with 0) of a string S
% A "clean" loop crosses one hole in one direction and returns back immediately
% after. For instance [...,-r,+r,...]
% T - original state; TN - final state
% We assume that the N-th crossing is a loop (otherwise, the action is not executable)

do_pull(S,N,T,TN) :-
    member(chain(S)=Zs,T),
    append(Zs0,[(X:A),(_:B)|Zs1],Zs), length(Zs0,N), % split Zs

    % Get all loops on string S
    findall(SLoop,(member(SLoop,T),loop_string(SLoop,S)),SLoops),

    % Crossings A B on S will disappear.
    % Find out which loops disappear:
    %this will include l(S,X,A,B), the pulled loop
    findall(L,(member(L,SLoops),has_some_xing(L,[A,B])),RemovLs),
    % remove them from the list of SLoops
    remove_all(RemovLs,SLoops,SLoops1),
    loop_replacements(SLoops1,RemovLs,Rs), % Rs are replacements L -> L2
    member((l(S,X,A,B) -> L2),Rs),
    opposite(X,Y),
    % find all the strings crossing l(S,X,A,B) and pick them
    (retractall(newxing(_)),!;true),
    change_fluents(pulled_pick,(l(S,X,A,B),L2,Y),T,T2),
    findall(NX,newxing(NX),NewXings),

    % Add all the new crossings to the pulled string
    % (new ids between A and B are assigned)
    % The ordering among those xings is not determined.
    new_ids(NewXings,A,B,NewIdXings),
    append([Zs0,NewIdXings,Zs1],Zs2),

    % Update chain(S)
    append(Pre,[chain(S)=_|Suf],T2),
    append(Pre,[chain(S)=Zs2|Suf],T3),

    % Replace removed loops (they may inherit a bigger loop)
    change_fluents(replace_loops,Rs,T3,T4),

    % Finally, recompute loops accordingly to new crossings
    update_loops(T4,TN),
    true.

new_ids([],_,_,[]):-!.
new_ids([X|Xs],A,B,[(X:C)|Zs]) :- midpoint(A,B,C), new_ids(Xs,C,B,Zs).
```

```
% Decides whether a loop on string S refers to some of the crossings in Xs
has_some_xing(l(_,_,A,B),Xs):-member(A,Xs),!;member(B,Xs).

% Check whether loop A-B is strictly included (starts,during,finishes) loop C-D
% provided that X and Y are on the "same side"
included(l(S,X,A,B),l(S,Y,C,D)) :- same_side(X,Y),A>=C, D>=B, (A \= C,!; B \= D).
same_side(- _,- _).
same_side(+ _,+ _).

loop_string(l(S,_,_,_),S).

% next_super_loop(T,L,T2)
% Look for the smallest loop Ln in set of loops Ls for same
% string S that "includes" loop L for S
next_super_loop(Ls,L,Ln) :-
    member(Ln,Ls), included(L,Ln),
    \+ (member(L3,Ls), included(L,L3), included(L3,Ln)).

% For each removed loop, add its next super loop (if existing).
% Otherwise add "none"
loop_replacements(_,[],[]) :- !.
loop_replacements(SLoops,[L|Ls],[(L->L2) | Rs]) :-
    (next_super_loop(SLoops,L,L2),!; L2=none),loop_replacements(SLoops,Ls,Rs).

% pulled_pick(chain(S)=Zs, (L,L2,Y), ZsN)
% Replace all crossings of loop L by L2 (or remove if L2=none)
% and create, before and after, new loops crossing towards Y
% The pulled string will cross those loops in the same direction
% each crossed new loop NX is asserted as a fact newxing(NX)
% pulled_pick is designed to be used by change_fluents

pulled_pick(chain(S)=Zs,(L,L2,Y),ZsN) :-
    append(Zs0,[(X:A)|Zs1],Zs),xing_sign_hole(X,D,L),!,
                                    % we found a crossing thru loop L
    length(Zs0,N),                  % locate its position
    N1 is N-1,
    append(Zs0,Zs1,Zs2),            % Zs2 = result of removing the crossing
    pick(S,Zs2,N1,Y,NewLoop,A,Zs3), % make new pick towards Y, using midpoint A
    xing_sign_hole(NX,D,NewLoop),
    opposite(NX,NX1),
    asserta(newxing(NX1)),

    ( L2=none,!,Zs4=Zs3             % if L2=none then crossing is removed
    ; xing_sign_hole(W,D,L2),       % otherwise, create a crossing through L2
      N2 is N+2,
      insert((W:A),N2,Zs3,Zs4)      % and insert it in the middle of the new pick
    ),
    pulled_pick(chain(S)=Zs4,(L,L2,Y),ZsN).  % continue making replacements
pulled_pick(chain(_S)=Zs,(_,_,_),Zs).

% replace_loops(chain(S)=Zs,Rs,ZsN)
% Rs provides a list of loop replacements L -> L2 that must be done on Zs

replace_loops(chain(_S)=[],_Rs,[]) :- !.
replace_loops(chain(S)=[(X:A) | Zs],Rs,ZsN) :-
      xing_sign_hole(X,D,L), member((L -> L2),Rs),!,
      ( L2=none,!,replace_loops(chain(S)=Zs,Rs,ZsN)  % This loop is removed
      ; xing_sign_hole(Y,D,L2), replace_loops(chain(S)=Zs,Rs,Zs2), ZsN=[(Y:A)|Zs2]
      ).
replace_loops(chain(S)=[Z|Zs],Rs,[Z|ZsN]) :-replace_loops(chain(S)=Zs,Rs,ZsN).

get_loop_id(T0,X,W) :-
    xing_sign_hole(X,D,loop(S2,M,N)),
    member(chain(S2)=Zs,T0),
    nth0(M,Zs,(Y:A),_),
    nth0(N,Zs,(_:B),_),
    xing_sign_hole(W,D,l(S2,Y,A,B)).

% Some predicates about crossings (xing) and holes
opposite(+X,-X).
opposite(-X,+X).

xing_hole(+X,X).
xing_hole(-X,X).
xing_sign_hole(+X,+,X).
xing_sign_hole(-X,-,X).

% Removing from a list
remove_all(Xs,L,L2) :- partition(contains(Xs),L,_,L2).
contains(L,X) :- member(X,L).

% Insert X at position N
insert(X,N,Ls,Ls2) :-
    N1 is N-1, append(Pre,Suf,Ls), length(Pre,N1), append(Pre,[X|Suf],Ls2).

write_state(T) :- repeat, (member(F,T),write(F),nl,fail; !).

writeall(Xs):- repeat, (member(X,Xs),write(X),fail; !).
```