Generating Readable Proofs: A Heuristic Approach to Theorem Proving With Spider Diagrams

Jean Flower, Judith Masthoff, and Gem Stapleton

The Visual Modelling Group
University of Brighton, Brighton, UK
www.cmis.brighton.ac.uk/research/vmg
{J.A.Flower,Judith.Masthoff,G.E.Stapleton}@bton.ac.uk

Abstract. An important aim of diagrammatic reasoning is to make it easier for people to create and understand logical arguments. We have worked on spider diagrams, which visually express logical statements. Ideally, automatically generated proofs should be short and easy to understand. An existing proof generator for spider diagrams successfully writes proofs, but they can be long and unwieldy. In this paper, we present a new approach to proof writing in diagrammatic systems, which is guaranteed to find shortest proofs and can be extended to incorporate other readability criteria. We apply the A^* algorithm and develop an admissible heuristic function to guide automatic proof construction. We demonstrate the effectiveness of the heuristic used. The work has been implemented as part of a spider diagram reasoning tool.

1 Introduction

In this paper, we show how readability can be taken into account when generating proofs in a spider diagram reasoning system. In particular, we show how the A* heuristic search algorithm can be applied.

If diagrammatic reasoning is to be practical, then tool support is essential. Proof writing in diagrammatic systems without software support can be time-consuming and error prone.

In [4] we present a proving environment that supports reasoning with spider diagrams. This proving environment incorporates automated proof construction: an algorithm generates a proof that one spider diagram entails another, provided such a proof exists. This algorithm usually produces long and somewhat unwieldy proofs. These unnatural proofs suffice if one only wishes to know that there *exists* a proof. However, if one wishes to *read* and *understand* a proof then applying the algorithm may not be the best approach to constructing a proof.

Readability and understandability of proofs are important because they:

– support *education.* The introduction of calculators has not stopped us teaching numeracy. Similarly, automatic proof generators will not stop us teaching students how to construct proofs manually. An automated proof generator can support the learning process, but only if it generates proofs similar to those produced by expert humans [22].

– increase *notational understanding*. People will need to write and verify the specifications (premise and conclusion) given to the automated proof generator. Specification is the phase where most mistakes can be introduced [11]. A good understanding of the notation used is therefore needed. Reading (and writing) proofs helps to develop that understanding.
– provide *trust* in generated proofs. As discussed in [11], automated proof generators can themselves contain flaws[1] and many people remain wary of proofs they cannot themselves understand [22].
– encourage *lateral thinking*. Understanding a proof often leads to ideas of other theorems that could be proved.

Research has been conducted on how to optimize the *presentation* of proofs, optimizing the layout of proofs or annotating proofs with or even translating them into natural language (see for example [15],[13],[1]). The diagrammatic reasoning community attempts to make reasoning easier by using diagrams instead of formulas. For instance, we are working on presenting proofs as sequences of diagrams.

In this paper, we enhance the proving environment presented in [4] by developing a heuristic approach to theorem proving in the spider diagram system. We define numerical measures that indicate 'how close' one spider diagram is to another and these measures guide the tool towards rule applications that result in shorter and, therefore, hopefully more 'natural' proofs. In fact, the method is guaranteed to find a shortest proof, provided one exists. Note that heuristic approaches have been used in automated theorem provers before, but mainly to make it faster and less memory intensive to find a proof. This is a nice side effect of using heuristics, but our main reason is readability of the resulting proof. Often heuristics used are not numerical such as 'in general, this rule should be applied before that rule' (e.g. [12]).When numerical heuristics have been used (e.g. [5]), they have usually been used to find a proof quickly rather than to optimize the proofs.

Many other diagrammatic reasoning systems have been developed, see [9] for an overview. For example, the DIAMOND system allows the construction of diagrammatic proofs, learning from user examples. The kinds of diagrams considered are quite different to spider diagrams.

In [18] Swoboda proposes an approach towards implementing an Euler/Venn reasoning system, using directed acyclic graphs (DAGs). The use of DAGs to compare diagrams was the focus of [19] in which two diagrams are compared to assess the correctness of a single diagram transformation. It is possible that the work on DAGs, if extended to assess a sequence of diagram transformations, could show some similarity to the measures we give in section 4. The existing DAGs work does not create proofs but rather checks the validity of a proof candidate.

The structure of this paper is as follows. In Section 2, we introduce a simplified (unitary diagrams only) version of our spider diagram reasoning system. In

[1] If an algorithm has been proved correct, still the correctness proof or the implementation could contain errors.

Section 3, we discuss the A* heuristic search algorithm that we have applied to this reasoning system. In Section 4, we describe a so-called admissible heuristic function that measures the difference between two spider diagrams. In Section 5, we briefly discuss how the heuristic diagrammatic proof generator has been implemented, and evaluate its results. We conclude by discussing how the cost element of A* can be used to further enhance understandability (the shortest proof is not necessarily the easiest to understand), how the heuristic can be used to support interactive proof writing, and what we expect from extending this approach to non-unitary spider diagrams and so-called constraint diagrams.

2 Unitary spider diagrams

In this section, we briefly and informally introduce our diagrammatic reasoning system. For reasons of clarity, we restrict ourselves in this paper to so-called *unitary* spider diagrams and the reasoning rules associated with those. We will briefly return to non-unitary diagrams in Section 6.

Simple diagrammatic systems that inspired spider diagrams are Venn and Euler diagrams. In Venn diagrams all possible intersections between contours must occur and shading is used to represent the empty set. Diagram d_1 in Fig. 1 is a Venn diagram. Venn-Peirce diagrams [14] extend the Venn diagram notation, using additional syntax to represent non-empty sets. Euler diagrams exploit topological properties of enclosure, exclusion and intersection to represent subsets, disjoint sets and set intersection respectively. Spider diagrams are based

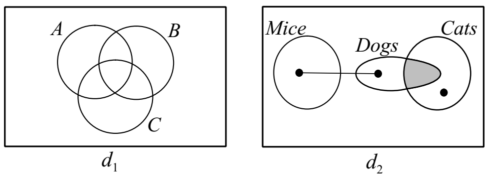

Fig. 1. A Venn diagram and a spider diagram.

on Euler diagrams. *Spiders* are used to represent the existence of elements and *shading* is used to place upper bounds on the cardinalities of sets. A spider is drawn as a collection of dots (the *feet*) joined by lines. The diagram d_2 in Fig. 1 is a spider diagram. Sound and complete reasoning rules for various spider diagram systems have been given, e.g. [4].

2.1 Syntax and semantics of unitary spider diagrams

In this section, we will give an informal description of the syntax and semantics of unitary spider diagrams. Details and a formal description can be found at [17].

- A **contour** is a labelled shape in the diagram used to denote a set.
- A **boundary rectangle** is an unlabelled rectangle that bounds the diagram and denotes the universal set.

– A **zone**, roughly speaking, is a bounded area in the diagram having no other bounded area contained within it. More precisely, a zone can be described by the set of labels of the contours that contain it (the containing label set) and the set of labels of the contours that exclude it (the excluding label set). A zone denotes a set by intersection and subtraction of the sets denoted by the contours that contain it and the contours that exclude it respectively.
– A **spider** is a tree with nodes, called **feet**, placed in different zones. A spider **touches** a zone if one of its feet appears in that zone. The set of zones a spider touches is called its **habitat**. A spider denotes the existence of an element in the set represented by its habitat. Distinct spiders represent the existence of distinct elements.
– A zone can be **shaded**. In the set represented by a shaded zone, all of the elements are represented by spiders. So, a shaded zone with no spiders in it represents the empty set.
– A **unitary diagram** is a finite collection of contours (with distinct labels), shading and spiders properly contained by a boundary rectangle.

The unitary diagram d_2 in Fig. 1 contains three labelled contours and five zones, of which one is shaded. There are two spiders. The spider with one foot inhabits the zone inside (the contour labelled) $Cats$, but outside $Dogs$ and $Mice$. The other spider inhabits the zone set which consists of the zone inside $Mice$ and the zone inside $Dogs$ but outside $Cats$. The diagram expresses the statement "no mice are cats or dogs, no dogs are cats, there is a cat and there is something that is either a mouse or a dog". A semantically equivalent diagram could be drawn which presents the contours $Dogs$ and $Cats$ as disjoint.

The semantics for spider diagrams is model-based. A model assigns sets to diagram contours, and zones to appropriate intersections of sets (and their complements). Zones which are absent from a diagram (for example if two contours are drawn disjoint) correspond to empty sets. Spiders assert the existence of (distinct) elements in the sets and shading (with spiders) asserts an upper bound on the cardinality of the sets. A zone which is shaded but untouched by spiders corresponds to the empty set in any model, but a zone which has a single-footed spider and is shaded corresponds to a set with exactly one element. A full description of the semantics can be found at [17].

2.2 Reasoning Rules

In this section we will give informal descriptions of the reasoning rules for unitary spider diagrams. Each rule is expressed as a transformation of one unitary spider diagram into another. Formal descriptions can be found at [21].

Rule 1 ***Add contour.*** *A new contour can be added. Each zone is split into two zones (one inside and one outside the new contour) and shading is preserved. Each foot of a spider is replaced by a connected pair of feet, one in each of the two new zones.* For example, in Fig. 2, diagram d_2 is obtained from d_1 by adding the contour labelled B.

Rule 2 ***Delete contour.*** *A contour can be deleted. If, as a result, a spider has two feet in the same zone, these feet are contracted into a single foot.*

Rule 3 ***Add shaded zone.*** *A new, shaded zone can be added.*

Rule 4 ***Delete shaded zone.*** *A shaded zone that is not part of the habitat of any spider can be deleted.* For example, in Fig. 2, diagram d_3 is obtained from d_2 by deleting a shaded zone.

Rule 5 ***Erase shading.*** *Shading can be erased from any zone.*

Rule 6 ***Delete spider.*** *A spider whose habitat is completely non-shaded can be deleted.*

Rule 7 ***Add spider foot.*** *A spider foot can be added to a spider in a zone it does not yet touch.*

With the semantics as sketched in the previous section, each of these rules has been proven to be sound. This means that an application of any rule to a diagram yields a second diagram representing a semantic consequence of the first diagram. A sequence of diagrams and rule applications also gives a semantic consequence, and in this logic system, is a proof.

Let d_1 and d_2 be diagrams. We say d_2 is **obtainable** from d_1, denoted $d_1 \vdash d_2$, if and only if there is a sequence of diagrams $\langle d^1, d^2, ..., d^m \rangle$ such that $d^1 = d_1$, $d^m = d_2$ and, for each k where $1 \leq k < m$, d^k can be transformed into d^{k+1} by a single application of one of the reasoning rules. Such a sequence of diagrams is called a **proof** from **premise** d_1 to **conclusion** d_2.

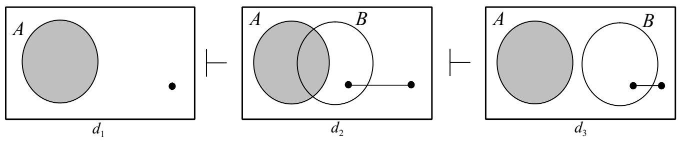

Fig. 2. Applications of Add Contour and Delete Shaded Zone.

3 A* applied to proof writing

To construct a proof, a rule needs to be applied to the premise diagram, followed by another rule to the resulting diagram, and so on, until the conclusion diagram is reached. At any stage in the proof, multiple rules might be applicable. For instance, in Figure 2, many rules can be applied to diagram d_1, such as Add Contour B, Delete Contour A, Delete Spider, Erase Shading, and Add Spider Foot. Not all rules applications help to find a shortest proof (e.g., only applying Add Contour B to d_1 would help to find a shortest proof to d_3), and some rule applications might even make it impossible to find a proof (e.g., applying Delete Spider to d_1 results in a diagram from which d_3 can no longer be proven). Human proof writers might intelligently choose the next rule to apply, in order to reach the conclusion diagram as quickly as possible. The problem of deciding which rules to apply to find a proof is an example of a more general class of so-called search problems, for which various algorithms have been developed (see [10] for an overview). Some of these algorithms, so-called blind-search algorithms, systematically try all possible actions. Others, so-called best-first search algorithms, have been made more intelligent, and attempt (like humans) to intelligently choose which action to try first. A^* is a well known best-first search algorithm [6].

A^* stores an ordered sequence of proof attempts. Initially, this sequence only contains a zero length proof attempt, namely the premise diagram. Repeatedly, A^* removes the first proof attempt from the sequence and considers it. If the last diagram of the proof attempt is the conclusion diagram, then an optimal proof has been found. Otherwise, it constructs additional proof attempts, by extending the proof attempt under consideration, applying rules wherever possible to the last diagram.

The effectiveness of A^* and the definition of "optimal" is dependent upon the ordering imposed on the proof attempt sequence. The ordering is derived from the sum of two functions. One function, called the **heuristic**, estimates how far the last diagram in the proof attempt is from the conclusion diagram. The other, called the **cost**, calculates how costly it has been to reach the last diagram from the premise diagram. We define the cost of applying a reasoning rule to be one. So, the cost is the number of reasoning rules that have been applied to get from the premise diagram to the last diagram (i.e. the length of the proof attempt). The new proof attempts are inserted into the sequence, ordered according to the cost plus heuristic.

A^* has been proven to be complete and optimal, i.e. always finding the best solution (because all reasoning rules have cost 1, this means the shortest proof), if one exists, provided the heuristic used is **admissible** [2]. A heuristic is admissible if it is **optimistic**, which means that it never overestimates the cost of getting from a premise diagram to a conclusion diagram. As all reasoning rules have cost equal to one, this means that the heuristic should give a lower bound on the number of proof steps needed in order to reach the conclusion diagram.

The amount of memory and time needed by A^* depends heavily on the quality of the heuristic used. For instance, a heuristic that is the constant function zero is admissible, but will result in long and memory-intensive searches. The better the heuristic (in the sense of accurately predicting the length of a shortest proof), the less memory and time are needed for the search. In this paper, we present a highly effective heuristic for A^* applied to proof writing in a unitary spider diagram reasoning system.

4 Heuristic function for unitary diagrams

As discussed above, in order to apply A^* we need a heuristic function that gives an optimistic (admissible) estimate of how many proof steps it would take to get from a premise diagram to the conclusion diagram. In this section, we develop various measures between two unitary diagrams. The overall heuristic we use is built from these measures. The heuristic gives a lower bound on the number of proof steps required and is used to help choose rule applications when constructing proofs in our implementation.

There are four types of differences that can be exhibited between two unitary diagrams: between the contour labels, the zones, the shaded zones and the spiders.

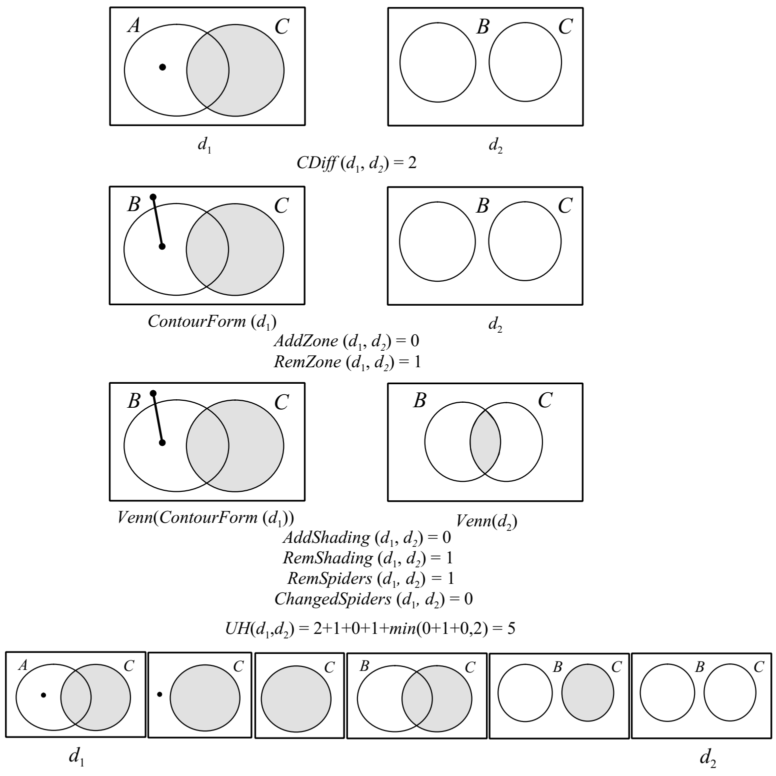

Fig. 3. Calculating the heuristic: example 1.

4.1 The contour difference measure

The only rules that alter a unitary diagram's contour set are Add Contour and Delete Contour, which respectively add and delete one contour at a time. Thus we define the **contour difference measure** between diagrams d_1 and d_2 to be the size of the symmetric difference of the label sets $Cont(d_1)$ (the set of labels of the contours of d_1) and $Cont(d_2)$:

$$CDiff(d_1, d_2) = |Cont(d_2) - Cont(d_1)| + |Cont(d_1) - Cont(d_2)|$$

The contour difference between d_1 and d_2 in Figure 3 is given by

$$CDiff(d_1, d_2) = |\{A, C\} - \{B, C\}| + |\{B, C\} - \{A, C\}| = 2$$

whereas the contour sets in Figure 4 are equal, so $CDiff(d_1, d_2) = 0$.

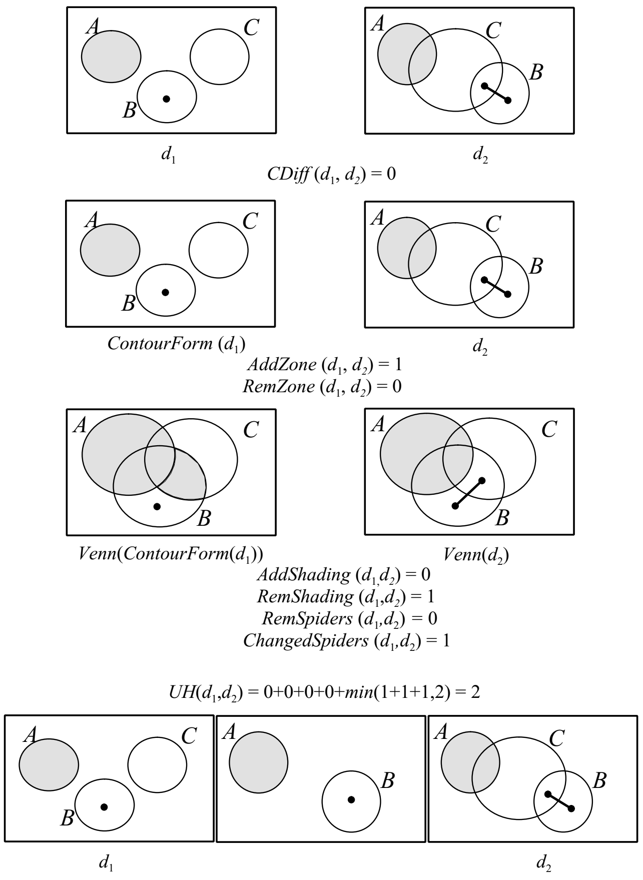

Fig. 4. Calculating the heuristic: example 2.

4.2 The zone difference measures

A diagram's zone set can be altered by the rules Add Contour, Delete Contour, Add Shaded Zone, and Delete Shaded Zone. The calculation of $CDiff$ identified contour differences between d_1 and d_2, and Add and Delete Contour rules would have to appear in any proof from d_1 to d_2 to fix these differences. We need to determine whether these rules are sufficient to account for changes in the zone sets, or whether other applications of rules to add or delete shaded zones are needed. In order to calculate the zone difference measures for the heuristic, take d_1 and apply the relevant Delete Contour and Add Contour rules to make a new diagram, $ContourForm(d_1, d_2)$ which has the same contour set as d_2. If the zone sets of $ContourForm(d_1, d_2)$ and d_2 don't match, we may also need to use Add Shaded Zone and/or Delete Shaded Zone in any proof from d_1 to d_2.

Two zones are deemed **equal** if they have the same set of containing contour labels and the same set of excluding contour labels. Define the two **zone**

difference measures between diagrams d_1 and d_2 to be

$$AddZone(d_1, d_2) = \begin{cases} 1 \text{ if } Zones(d_2) \not\subseteq Zones(ContourForm(d_1, d_2)) \\ 0 \text{ otherwise} \end{cases}$$

$$RemZone(d_1, d_2) = \begin{cases} 1 \text{ if } Zones(ContourForm(d_1, d_2)) \not\subseteq Zones(d_2) \\ 0 \text{ otherwise.} \end{cases}$$

The sum $CDiff(d_1, d_2) + AddZone(d_1, d_2) + RemZone(d_1, d_2)$ is an optimistic estimate of the number of applications of Add Contour, Delete Contour, Add Shaded Zone and Delete Shaded Zone which are required in a proof from d_1 to d_2.

In Figure 3, the diagram $ContourForm(d_1)$ has a zone not present in d_2 (the zone in both contours B and C), so $RemZone(d_1, d_2) = 1$. All zones in d_2 are present in $ContourForm(d_1)$, so $AddZone(d_1, d_2) = 0$. The sum of measures is three, and this corresponds to three rule applications: Remove Contour (A), Add Contour (B) and Delete Shaded Zone (intersection of B and C) which have to be present in any proof from d_1 to d_2. In Figure 4, diagram d_2 has two zones not in $ContourForm(d_1)$: the zone in A and C and the zone in B and C. Although there are two extra zones, we still find that $AddZone(d_1, d_2) = 1$. The sum above, in this case, is $0 + 1 + 0 = 1$ and any proof from d_1 to d_2 indeed requires at least one step.

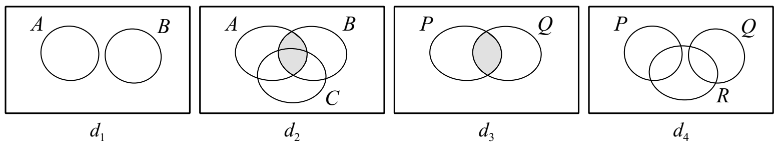

Fig. 5. Examples to justify capping measures at one.

From these examples alone, the limiting of $AddZone$ and $RemZone$ to a maximum value of one may seem counterintuitive. In Figure 5, diagrams d_1 and d_2 have $CDiff(d_1, d_2) = 1$ and $AddZone(d_1, d_2) = 1$. Because there are two zones in d_2 which are not present in $ContourForm(d_1, d_2)$, one might conclude that two applications of Add Shaded Zone would be required to transform d_1 into d_2, and a suitable $AddZone$ measure would be two. However, we can add just one shaded zone to d_1 before adding the contour C. For this reason, the $AddZone$ measure is capped at a value of 1, even where $ContourForm(d_1, d_2)$ has more than one zone that is not in d_1. Diagrams d_3 and d_4 in Figure 5 similarly justify the capping of the $RemZone$ measure at one.

4.3 The shading difference measures

So far, we have captured differences between contour sets and zone sets. We also want a way to compare the shading between diagrams. The shading can not only be affected by the rule Erase Shading, but also by any of the rules required to make the contour sets and the zone sets the same.

To measure the difference in shading between d_1 and d_2, we take the diagram $ContourForm(d_1, d_2)$ and add shaded zones until the diagram is in Venn form (every possible zone is present, given the contour label set), giving $Venn(ContourForm(d_1, d_2))$. We also add shaded zones to d_2 until d_2 is in Venn form, giving $Venn(d_2)$. The **shading difference measures** between diagrams d_1 and d_2 are defined to be

$$AddShading(d_1, d_2) = \begin{cases} \infty \text{ if } ShadedZones(Venn(d_2)) \\ \quad \not\subseteq ShadedZones(Venn(ContourForm(d_1, d_2))) \\ 0 \ \ \text{otherwise} \end{cases}$$

$$RemShading(d_1, d_2) = \begin{cases} 1 \text{ if } ShadedZones(Venn(ContourForm(d_1, d_2))) \\ \quad \not\subseteq ShadedZones(Venn(d_2)) \\ 0 \text{ otherwise.} \end{cases}$$

The allocation of ∞ as the *AddShading* measure indicates that there is no proof from d_1 to d_2. In the examples shown in Figures 3 and 4, a proof does exist between d_1 and d_2, and in both cases $AddShading(d_1, d_2) = 0$. Also, in both figures, $Venn(ContourForm(d_1, d_2))$ has more shading than $Venn(d_2)$, so $RemShading(d_1, d_2) = 1$. The value of *RemShading* is capped at 1 for similar reasons to the capping of *AddZone* and *RemZone*.

The sum $CDiff + AddZone + RemZone + AddShading + RemShading$ is an optimistic estimate of the number of applications of Add Contour, Delete Contour, Add Shaded Zone, Delete Shaded Zone, and Erase Shading which are required in a proof from d_1 to d_2 (i.e. the sum is a lower bound on the number of proof steps required).

4.4 The spider difference measures

The rules Delete Spider and Add Spider Foot change the number of spiders in a diagram and the habitats of spiders, respectively. In addition, deleting and reintroducing a contour can also affect the habitats of spiders.

We can see that no rules introduce spiders, so if d_2 has more spiders than d_1 then we define the spider difference measure to be infinite (in effect, blocking the search for proofs between such diagrams). If we have fewer spiders in d_2 than d_1, then the rule Delete Spider must have been applied and the difference between the numbers of spiders contributes to the heuristic.

If a spider in d_2 has a different habitat to all spiders in $ContourForm(d_1, d_2)$ (i.e. it's *unmatched* in d_1) then some rule must be applied to change the habitat of a spider in d_1 to obtain the spider in d_2. That rule could be Add Spider Foot. Alternatively, the deletion and reintroduction of a contour can change many spiders' habitats. We say that n spiders in d_2 are **unmatched** in d_1 if the bag subtraction Bag(habitats of spiders in d_2) $-$ Bag(habitats of spiders in $ContourForm(d_1, d_2)$) has n elements. Define

$$RemSpiders(d_1, d_2) = \begin{cases} \infty \quad \text{if } NumSpiders(d_1) - NumSpiders(d_2) < 0 \\ NumSpiders(d_1) - NumSpiders(d_2) \quad \text{otherwise} \end{cases}$$

$$ChangedSpiders(d_1, d_2) = \text{Number of spiders in } d_2 \text{ unmatched in } d_1.$$

In Figure 3, there is one spider in d_1 and none in d_2, so one application of the Delete Spider rule is required in any proof from d_1 to d_2. The measure $RemSpiders(d_1, d_2) = 1$, but there are no unmatched spiders in d_2, so $ChangedSpiders(d_1, d_2) = 0$. In Figure 4, d_1 and d_2 have the same number of spiders, so $RemSpiders(d_1, d_2) = 0$. However, the spider in d_2 is unmatched in d_1 so $ChangedSpiders(d_1, d_2) = 1$.

4.5 The unitary diagram heuristic

We have built seven measures by considering possible differences between the premise and conclusion diagrams. To give us a lower bound on the length of a proof with premise d_1 and conclusion d_2 we take the sum of the measures described above, but limit the contributions from *AddZone*, *RemShading* and *ChangedSpiders* to 2. This is because zones, shading and spiders can change by applications of Delete Contour followed by Add Contour, as illustrated in Figure 4. Unless we cap the heuristic as shown, it will fail to be admissible, as required by the A* algorithm. Define the **unitary diagram heuristic** between d_1 and d_2 to be the sum

$$\begin{aligned} UH(d_1, d_2) = \quad & CDiff(d_1, d_2) + RemZone(d_1, d_2) \\ & + AddShading(d_1, d_2) + RemSpiders(d_1, d_2) \\ & + min \begin{cases} AddZone(d_1, d_2) + RemShading(d_1, d_2) \\ \quad +ChangedSpiders(d_1, d_2) \\ 2. \end{cases} \end{aligned}$$

Lemma 1. *Let d_1 and d_2 be unitary diagrams. If $UH(d_1, d_2) = \infty$ then $d_1 \nvdash d_2$.*

Theorem 1. *Let d_1 and d_2 be unitary diagrams. If there is a proof with premise d_1 and conclusion d_2 then that proof has length at least $UH(d_1, d_2)$. That is, the unitary diagram heuristic is optimistic.*

The proof strategy uses induction on lengths of proofs. Given a shortest proof length n between d_1 and d_2, let d_{next} be the second diagram in the proof. The proof from d_{next} to d_2 is length $n-1$ and so by induction, $n-1 \geq UH(d_{next}, d_2)$. We consider the seven rules in turn which could have been applied to d_1 to obtain d_{next}. In each case, we find a relationship between $UH(d_1, d_2)$ and $UH(d_{next}, d_2)$ which allows us to deduce that $n \geq UH(d_1, d_2)$.

5 Implementation and Evaluation

The heuristic approach to proof-finding has been implemented in java as part of a proving tool (available at [20]). The user can provide diagrams and ask the prover to seek a proof from one diagram to another. The tool allows the user to give a restriction on the rule-set used. Moreover, in order to assess the benefits gained from the heuristic defined in section 4, the user can choose between the *zero heuristic* (ZH) and the *unitary diagram heuristic* (as defined above). The zero heuristic simply gives $ZH(d_1, d_2) = 0$, for any diagrams d_1 and d_2. The A^* algorithm, when implemented with the *zero* heuristic, simply performs an inefficient breadth-first search of the space of all possible proof attempts (given that we have assumed the cost of each rule application to be 1).

Both heuristics succeed in finding proofs - the zero heuristic taking longer than the unitary diagram heuristic. The application records how many proof attempts were made during the search. This number can be thought of as a memory and time burden. The savings made using our heuristic, over using the zero heuristic, can be seen by comparing the number of proof attempts. In an extreme case, during the data collection described below, the zero heuristic required 70795 proof attempts while our heuristic only required 543 proof attempts.

We generated a random sample (size $n = 2400$) of pairs of unitary diagrams, d_1 and d_2, for which $d_1 \vdash d_2$. These diagrams had at most three contours, two spiders and had a shortest proof length of four steps between them (these choices were arbitrary - similar results can be obtained by using different data sets). For each pair, we recorded the number of proof attempts each heuristic took to find a shortest proof from d_1 to d_2. Since we are interested in the proportional saving, we calculated the ratio $\frac{n_1}{n_2}$ where n_1 is the number of proof attempts for the unitary diagram heuristic and n_2 is the number of proof attempts for the zero heuristic. A histogram showing the ratios obtained and their frequencies can be seen in Figure 6, as can a scatter plot of the raw data.

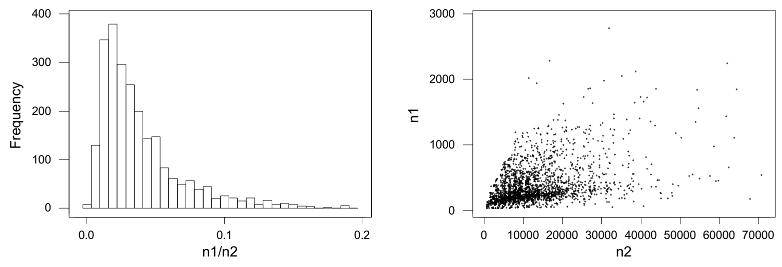

Fig. 6. Histogram showing the frequencies for each ratio and a scatter plot.

We found that the unitary diagram heuristic takes, on average, less than 4% of the number of proof attempts that the zero heuristic takes.

The A^* search algorithm was implemented in two ways - one stopping condition guarantees that one shortest proof was found (as discussed above), and a stronger stopping condition guarantees that all shortest proofs were found. This second stopping condition was implemented so that the data collected was not affected by the order in which rules were applied in the search process. Even though it's slower, the collection of all shortest proofs (instead of just one proof) could be of value in terms of maximizing readability of the outcome. Using the stronger stopping condition, the unitary heuristic takes, on average, less than 1.5% of the number of proof attempts that the zero heuristic takes.

6 Conclusion and further work

In this paper, we have demonstrated how a heuristic A^* approach can be used to automatically generate optimal proofs in a unitary spider diagram reasoning system. We regard this as an important step towards generating readable proofs. However, our work has been limited in a number of ways.

The first limitation is that we have assumed the cost of applying each rule to be equal. This results in the optimal proofs being found by A^* being the shortest proofs. However, as already indicated in the introduction, the conciseness of a proof does not have to be synonymous with its readability and understandability. The cost element of the evaluation function can be altered to incorporate other factors that impact readability. For instance:

– Comprehension of rules. There may be a difference in how difficult each rule is to understand. This might depend on the number of side effects of a rule. For instance, Delete Spider only deletes a spider without any side effects, while Add Contour does not only add a new contour, but can also add new feet to existing spiders. This might make an Add Contour application more difficult to understand. In a training situation, a student might already be very familiar with some rules but still new to other rules. We can model a difference in the relative difficulty of rules by assigning different costs. As long as we keep the minimum cost equal to 1, this would not impact the admissibility of the heuristic.
– Drawability of diagrams. As discussed in [7] not all diagrams are drawable, subject to some well-formed conditions. These conditions were chosen to increase the usability of diagrams, for instance, the diagram with two contours which are super-imposed could be hard for a user to interpret, and there is no way of drawing that diagram without concurrent contours, or changing the underlying zone set. Ultimately, we want to present a proof as a sequence of well-formed diagrams with rule applications between them. A proof would be less readable if an intermediate diagram could not be drawn. We can model this by increasing the cost of a rule application if the resulting diagram is not drawable.

Empirical research is needed to determine which other factors might impact readability, what the relative understandability of the rules is and to what extent that is person dependent, and how to deal with non-drawable diagrams.

The second limitation is that we have restricted ourselves to discussing the case of *unitary* spider diagrams. To enhance the practical usefulness of this work, we will need to extend this first to so-called compound diagrams, and next to the much more expressive constraint diagrams. Compound diagrams are built from unitary diagrams, using connectives $\sqcup$ and $\sqcap$ to present disjunctive and conjunctive information. If D_1 and D_2 are spider diagrams then so are $D_1 \sqcup D_2$ ("D_1 or D_2") and $D_1 \sqcap D_2$ ("D_1 and D_2"). In addition to the reasoning rules discussed in Section 2.2 which operate on the unitary components, many reasoning rules exist that change the structure of a compound diagram [21]. We have started to investigate heuristic measures for compound diagrams. Our implementation (available on [20]) is capable of generating proofs for compound diagrams, using either the zero heuristic, or a simple heuristic that is derived only from the number of unitary components of the diagrams. Two rules were excluded, namely $D_1 \vdash D_1 \sqcup D_2$, and $False \vdash D_1$. We have defined modified versions of the unitary measures, which we expect to be able to integrate into an effective heuristic for compound diagrams, and we are exploring the use of additional structural information.

Constraint diagrams are based on spider diagrams and include further syntactic elements, such as *universal spiders* and *arrows*. Universal spiders represent universal quantification (in contrast, spiders in spider diagrams represent existential quantification). Arrows denote relational navigation. In [3] the authors give a reading algorithm for constraint diagrams. A constraint diagram reasoning system (with restricted syntax and semantics) has been introduced in [16]. Since the constraint diagram notation extends the spider diagram notation, this is a significant step towards the development of a heuristic proof writing tool for constraint diagrams. In [16], the authors show that the constraint diagram system they call CD1 is decidable. However there are more expressive versions of the constraint diagram notation that may or may not yield decidable systems. If a constraint diagram system does not yield a decidable system then a heuristic approach to theorem proving will be vital if we are to automate the reasoning process.

In addition to its use in automatic proof generation, our heuristic measure can also be used to support interactive proof writing (e.g. in an educational setting). It can advise the user on the probable implications of applying a rule (e.g. "If you remove this spider, then you will not be able to find a proof any more", or "Adding contour B will decrease the contour difference measure, so might be a good idea"). Possible applications of rules could be annotated with their impact on the heuristic value. The user could collaborate with the proof generator to solve complex problems. More research is needed to investigate how useful our heuristic measures are in an interactive setting.

Acknowledgment

Gem Stapleton thanks the UK EPSRC for support under grant number 01800274. Jean Flower was partially supported by the UK EPSRC grant GR/R63516.

References

1. B.I. Dahn and A. Wolf. Natural language presentation and combination of automatically generated proofs. Proc. *Frontiers of Combining Systems*, Muenchen, Germany, p175-192, 1996.
2. R. Dechter, and J. Pearl. Generalized best-first search strategies and the optimality of A^*. *Journal of the Association for Computing Machinery, 32*, 3, p505-536. 1985.
3. A. Fish, J. Flower, and J. Howse. A Reading algorithm for constraint diagrams. Proc. *IEEE Symposium on Visual Languages and Formal Methods*, New Zealand, 2003.

4. J. Flower, and G. Stapleton. Automated Theorem Proving with Spider Diagrams. Proc. *Computing: The Australasian Theory Symposium*, Elsevier science, 2004.
5. C. Goller. Learning search-control heuristics for automated deduction systems with folding architecture networks. Proc. *European Symposium on Artificial Neural Networks.* D-Facto publications, Apr 1999.
6. P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on System Science and Cybernetics*, 4, 2, p100-107, 1968.
7. J. Flower, and J. Howse. Generating Euler Diagrams. Proc. *Diagrams* p.61-75, Springer-Verlag, 2002.
8. J. Howse, F. Molina, and J. Taylor. SD2: A sound and complete diagrammatic reasoning system. Proc. *IEEE Symposium on Visual Languages (VL2000)*, IEEE Computer Society Press p.402-408, 2000.
9. M. Jamnik, Mathematical Reasoning with Diagrams, CSLI Publications, 2001.
10. G.F. Luger. Artificial intelligence: Structures and strategies for complex problem solving. Fourth Edition. Addison Wesley: 2002.
11. D. MacKenzie. Computers and the sociology of mathematical proof. Proc. *3rd Northern Formal Methods Workshop*, 1998. http://www1.bcs.org.uk/DocsRepository/02700/2713/mackenzi.pdf
12. S. Oppacher and S. Suen. HARP: A tableau-based theorem prover. *Journal of Automated Reasoning*, 4, p69-100, 1988.
13. F. Piroi and B. Buchberger. Focus windows: A new technique for proof presentation. In J. Calmet, B. Benhamou, O. Caprotti, L. Henocque, and V. Sorge, editors, Artificial Intelligence, Automated Reasoning and Symbolic Computation. Proceedings of *Joint AICS'2002 - Calculemus'2002 Conference*, Marseille, France, July 2002. Springer Verlag. http://www.risc.uni-linz.ac.at/people/buchberg/papers/2002-02-25-A.pdf [Accessed August 2003].
14. S.-J. Shin. The Logical Status of Diagrams. Camb. Uni. Press, 1994.
15. J. Schumann and P. Robinson. [] or SUCCESS is not enough: Current technology and future directions in proof presentation. Proc. *IJCAR workshop: Future directions in automated reasoning*, 2001.
16. G. Stapleton, J. Howse, and J. Taylor. A constraint diagram reasoning system. Proc. *International conference on Visual Languages and Computing*, Knowledge systems institute, pp 263-270, 2003.
17. G. Stapleton, J. Howse, J. Taylor and S. Thompson. What can spider diagrams say? Proc. *Diagrams* 2004.
18. N. Swoboda. Implementing Euler/Venn reasoning systems, In Diagrammatic Representation and Reasoning, Anderson, M., Meyer, B., and Oliver, P., eds, p.371-386 Springer-Verlag, 2001.
19. N. Swoboda and G. Allwein. Using DAG Transformations to Verify Euler/Venn Homogeneous and Euler/Venn FOL Heterogeneous Rules of Inference. *Electronic Notes in Theoretical Computer Science*, 72, No. 3 (2003).
20. The Visual Modelling Group website. www.cmis.brighton.ac.uk/research/vmg.
21. The Visual Modelling Group technical report on spider diagram reasoning systems at www.cmis.brighton.ac.uk/research/vmg/SDRules.html
22. W. Windsteiger. An automated prover for Zermelo-Fraenkel set theory in *Theorema.* LMCS, 2002.