Inter-diagrammatic Reasoning

Michael Anderson*
Computer Science Department
University of Hartford
Dana Hall 230
West Hartford, Connecticut 06117
U. S. A.

Robert McCartney
Computer Science and Engineering
University of Connecticut
260 Glenbrook Road
Storrs, Connecticut 06269-3155
U. S. A.

Abstract

Endowing a computer with an ability to reason with diagrams could be of great benefit in terms of both human-computer interaction and computational efficiency through explicit representation. To date, research in diagrammatic reasoning has dealt with intra-diagrammatic reasoning (reasoning with a single diagram) almost to the exclusion of inter-diagrammatic reasoning (reasoning with related groups of diagrams). We postulate a number of general inter-diagrammatic operators and show how such operators can be useful in various diagrammatic domains. We develop a heuristic in the domain of game notation, derive fingering information in the domain of musical notation, and infer new information from related cartograms.

1 Introduction

Humans possess a highly developed ability to reason with visual information such as diagrams. It has been shown that endowing a computer with such an ability could be of great benefit in terms of both human-computer interaction and computational efficiency through explicit representation [Larkin and Simon, 1987]. To date, research in diagrammatic reasoning has dealt with intra-diagrammatic reasoning [Chandrasekaran *et al,* 1993; Narayanan, 1992; Narayanan, 1993] almost to the exclusion of inter-diagrammatic reasoning.

Intra-diagrammatic reasoning can be defined as a process of inference realized by the application of various operators to a given single diagram (for example, see [Furnas, 1992]). *Inter-diagrammatic reasoning,* on the other hand, can be defined as a process of inference realized by the application of operators to groups of related diagrams (for example, see [Anderson, 1994; Anderson and McCartney, 1995]). Diagrams are related if they can be combined in ways that produce useful information. Two types of related diagram groups can be defined, sequences and suites. These differ in the way in which a single diagram within a group is related to other diagrams in the same group.

A *diagram sequence* [Anderson, 1994; Anderson and McCartney, 1995] will be defined as a meta-diagram composed of a number of diagrams arranged in an order that incorporates some manner of forward moving time. Often, each of the diagrams in the sequence can be considered a discrete snapshot of some continuous phenomenon described by the sequence as a whole. Examples of such diagram sequences include game notation, chordal musical notation, weather maps, assembly instructions, and instructions for a product's use.

A *diagram suite* will be defined as a meta-diagram composed of a number of diagrams that present different facets of a given entity at the same moment in time. Each of the diagrams of the suite can be considered as a view of the entity from some single, unique perspective. When combined, the suite can present the entity as a whole. Examples of such diagram suites include architectural renderings, anatomical drawings, and cartograms.

We present a syntax and semantics of inter-diagrammatic reasoning and then introduce a number of inter-diagrammatic operators and functions. Next, example uses of these operators and functions are provided in various domains. A brief discussion of related work follows and, finally, we offer our conclusions.

2 Diagram Syntax and Semantics

Most generally, we will syntactically define a *diagram* to be a tessellation of a planar area such that it is completely covered by atomic two dimensional regions or elements. The semantic domain will be defined as $\{v_0, \ldots v_1\}$ denoting an /' valued, additive gray scale incrementally increasing from a minimum value v_0,WHITE, to a maximum value v1, BLACK. Intuitively, the gray scale values correspond to a discrete set of transparent gray filters that, when overlaid, combine to create a darker filter to a maximum of BLACK.

3 Diagrammatic Operators and Functions

The following primitive unary operators, binary operators, and functions provide a set of basic tools to facilitate the process of inter-diagrammatic reasoning. We have striven

*Previously at Sacred Heart University, Fairfield, CT, U.S.A.

for simplicity and generality in the hope that these tools might be applicable in a wide variety of diagrammatic domains.

3.1 Unary operators

NOT, denoted ~*d,* is a unary operator taking a single diagram that returns a new diagram where each element is BLACK if the corresponding element in *d* is WHITE, and WHITE otherwise.

3.2 Binary operators

Binary operators take two diagrams, *d,* and d2, of equal dimension and tessellation and return a new diagram where each element has a new value that is some function of the two corresponding elements in the operands.

OR, denoted d_y v d_2, returns the *maximum* of each pair of elements. The maximum of two corresponding elements is defined as the element whose value is closest to BLACK.

AND, denoted d_y A $d_{2>}$ returns the *minimum* of each pair of elements. The minimum of two corresponding elements is defined as the element whose value is closest to WHITE.

OVERLAY, denoted d1, + d_2, returns the *sum* of each pair of elements. The sum of values of corresponding elements is defined as the sum of their respective values' subscripts.

PEEL, denoted *d,* - d_2, returns the *difference* of each pair of elements. The difference of values of corresponding elements is defined as the difference of their respective values' subscripts.

ASSIGNMENT, denoted d_1 <= d_2, modifies of, such that each element has the value of the corresponding element in d_2. (Note that non-diagrammatic assignment will be symbolized as := and the equality relation as =.)

3.3 Functions over diagrams

NONNULL, denoted NONNULL(d), is a one place Boolean function taking a single diagram that returns FALSE if all elements of *d* are WHITE else it returns TRUE.

3.4 Functions over sets of diagrams

ACCUMULATE, denoted ACCUMULATE *(d, ds, o),* is a three place function taking an initial diagram, *d,* a set of diagrams of equal dimension and tessellation, *ds,* and the name of a binary diagrammatic operator, *o,* that returns a new diagram which is the accumulation of the results of successively applying *o* to *d* and each diagram in *ds* .

MAP, denoted MAP(f, *ds),* is a two place function taking a function *f* and a set of diagrams of equal dimension and tessellation, *ds,* that returns a new set of diagrams comprised of all diagrams resulting from application of *f* to each diagram in *ds.*

FILTER, denoted FILTERS *ds),* is a two place function taking a Boolean function, *f* and a set of diagrams of equal dimension and tessellation, *ds,* that returns a new set of diagrams comprised of all diagrams in *ds* for which *f* returns TRUE.

CARDINALITY, denoted CARDINALITY(s), is a one place function taking a finite set that returns the number of elements in 5.

4 Example Domains

As example uses of the previously postulated inter-diagrammatic operators and functions, we 1) develop a heuristic for a game, 2) infer the correct fingering of a sequence of instrumental chord diagrams, and, 3) infer the quality of precipitation in a suite of cartograms.

4.1 Battleship

We have chosen the simple game of *Battleship* as one domain in which to test the diagrammatic operators and functions. The domain and its constraints are described first followed by a description of the inference goal and a detailed account of a working system based on this domain.

Battleship, Figure 1, is a game for two in which both players place *ships* (groups of two, three, or five contiguous blocks— a *block* being the atomic element of this domain) diagonally, horizontally, or vertically on a indexed, ten by ten grid. Each player then tries to *sink* the other player's ships by *shooting* them (marking all of the blocks comprising the ship) without ever seeing the grid on which they are placed. This feat is accomplished by the currently attacking player sending a *salvo* of shots (announcing the coordinates of seven blocks) and the other player providing a *damage report* that details the number of hits sustained by each of his/her ships but not the indices of each hit. The winner is the player who sinks the other player's ships first.

We are interested in applying the proposed diagrammatic operators and functions to the end of predicting the best shots a player might take given the progress of the game so far. To clarify this process, we will discuss it in terms of a subset of Battleship. The subset will consist of the game as described limited to only a single ship, namely the *battleship* (a five block group). Further, we will consider the placement of the battleship as being completely random. Within these constraints, it is possible to construct a diagram via the proposed diagrammatic operators that displays the entire set of possible battleship positions and, by simple inspection, discover the blocks that are most likely to be included in the battleship being sought. The intuition is that we would like to combine information from each possible configuration of a battleship onto a single diagram. The "darkness" of a given block in this diagram indicates the number of possible battleships it could be a part of and, hence, its likelihood of being a good candidate for a next shot. This is equivalent to numerically calculating the probabilities for each cell.

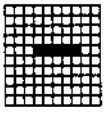


Figure 1: Battleship grid with single battleship

In the next section, we describe the process of displaying all possible positions of a ship on a single diagram. An example is then provided that details the diagrammatic reasoning required to develop a heuristic from a sequence of battleship shot boards.

Displaying a set

Figure 2 details the process by which an entire set of ships can be displayed on a single diagram. A diagram with only WHITE blocks (denoted D_1) is OVERLAYed with another diagram (denoted D_2). D_2 has one possible position of a battleship on it, represented as a contiguous five block region containing the first level of gray above WHITE (GRAY,). OVERLAYing produces another diagram that, after this single step, happens to be identical to D_2. This newly created diagram becomes the new D_1 which, in turn, is OVERLAYed with another diagram that has a different possible position of the battleship on it. Yet another diagram is produced that contains a representation of both of the ship positions so far included. This is due to the additive nature of the domain values previously defined combined with the semantics of the OVERLAY operator. The effect on the diagram is that ships that overlap each other make blocks they have in common darker than the blocks they don't have in common. In the example, the common blocks will now have the value of $GRAY_2$ (GRAY, + GRAY,) whereas the other blocks will have the value GRAY, (WHITE + GRAY,). This process is repeated until all possible positions of a ship have been overlayed onto a single diagram.

In more formal terminology, the function ACCUMULATE(0, *Suppositions,* +) is applied where 0 *(null diagram)* is the diagram initialized to WHITE and *ShipPositions* is a domain specific set of all diagrams of possible single battleship positions. In each of these diagrams, the blocks that are part of the battleship take the value v_1 while all other blocks take the value v_0. The final result of this application of ACCUMULATE is a diagrammatic representation of all possible ship positions with those blocks most likely to be included in a ship being the darkest and those that are least likely, the lightest.

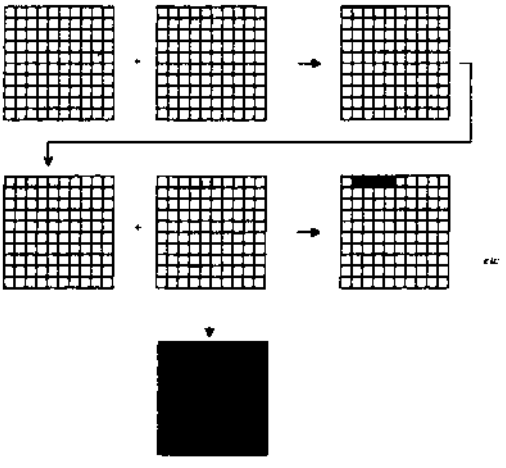


Figure 2: Overlaying battleships

Developing the heuristic

Figures 3 and 4 detail an example of the process by which the set of possible ships is constrained as the game progresses and a diagrammatic representation of the heuristically best shots is developed. First, a new salvo of shots is placed on the previous game board as BLACK blocks. At the start of the game, the board will contain only WHITE blocks but, as each turn is taken, this board will fill with shots from each turn. To differentiate between previous and new shots, the previous board (denoted *PreviousBoard)* is NEGATEd and ANDed with the current board (denoted *CurrentBoard)* giving the current salvo (denoted *CutrentSalvo)* of shots. Formally:

CurrentSalvo <= - *PreviousBoard* A *CurrentBoard*

Next, the number of hits (denoted *Hits)* scored by the salvo is determined. This is accomplished diagrammatically by ANDing the diagram containing the current salvo of shots with the diagram that contains the battleship in its actual position represented as GRAY1 blocks (denoted *Actual-Battleship).* Since ANDing is defined as taking the minimum of each block of a diagram, the resulting diagram will contain GRAY1 blocks for each hit in the current salvo and WHITE blocks everywhere else. Hits can then be counted, diagrammatically, by ANDing the resulting diagram with each member of a predefined set of *inspection diagrams* — diagrams used to isolate elements . In the current domain, the set of inspection diagrams is comprised of all possible diagrams containing a single BLACK block (denoted *Single-Blocks).* As each of these are ANDed with the diagram containing GRAY1 blocks for each hit in the current salvo, only those that have their single BLACK block in the element corresponding with the GRAY1 block will yield a new non-null diagram. The result of each operation is tested with the Boolean NONNULL function and its successes are counted thus producing the number of hits. The entire process can be formally, and more compactly, stated using the diagrammatic operators and functions *(X* is used in the standard way to denote function abstraction):

Hits :=

```
CARDINALITY(
  FILTER(NONNULL,
        MAP(λ(y) (y ∧ CurrentSalvo ∧ ActualBattleship),
            SingleBlocks)))
```

The First Salvo

In the example, the first salvo, Figure 3, results in no shot hitting the battleship as placed in Figure 1. This information is then reflected on a diagram by a process of overlaying similar to that previously described. Every possible instance of the battleship is overlayed as a contiguous five block region of GRAY1 onto a diagram initialized to WHITE. Now, however, a possible instance of a battleship must also conform to the number of hits specified. That is, in order to be considered possible, each five block region must overlap the number of hits (BLACK blocks on the current salvo diagram)

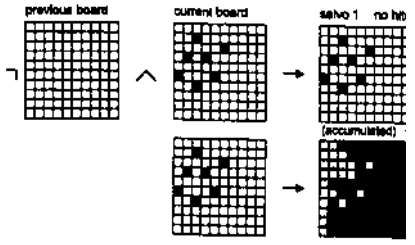


Figure 3: Battleship, the first salvo

exactly. In the current example, since there were no hits, a five block region that overlaps any BLACK block is *not* considered possible and, therefore, will not become part of the new set when displayed.

This effect is achieved by ANDing a given instance of a battleship, represented with GRAY1 blocks, with the diagram representing the current salvo. The resulting diagram will contain GRAY1 blocks for each hit that the given instance of a battleship overlaps and WHITE blocks everywhere else. These GRAY1 blocks are counted via the inspection diagrams as previously detailed and this count compared with the number of hits needed. If these numbers are equal, the given battleship instance is OVERLAYed on the accumulated result otherwise it is discarded. When all such battleship instances have been so OVERLAYed, the resulting diagram represents the current set of possible battleships newly constrained by the information in the damage report conveyed by the defending player. This process can be more formally stated using the diagrammatic operators and functions as:

```
ACCUMULATE(
    Ø,
    FILTER(λ(x) (Hits =
                CARDINALITY(
                  FILTER(NONNULL,
                         MAP(λ(y) ( y ∧ CurrentSalvo ∧ x),
                             SingleBlocks)))),
            ShipPositions),
)
```

The resulting diagram is a collection of blocks with values ranging from WHITE to BLACK. If the placement of the battleship is random, BLACK blocks are most likely to be contained in the battleship given the hit information so far with lighter shades of gray becoming decreasingly less likely. Further, given the damage report information, WHITE blocks are guaranteed not to be included as part of the battleship a player is seeking. This result (denoted *HeuristicDiagram),* then, can be considered a diagrammatic heuristic that indicates the probabilistically best shots for the next salvo.

The **Second Salvo**

The second salvo, Figure 4, uses the information previously derived by including within it the seven darkest blocks on the heuristic diagram. First, the previous board is NEGATEd and **ANDed** with the current board giving the current salvo of shots. The number of hits is determined as described

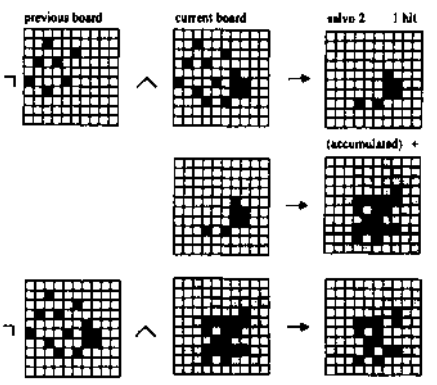


Figure 4: Battleship, the second salvo

previously and results in a count of one. The set of possible ships, *Suppositions,* is updated so as not to include any ship instances that were deemed impossible by the previous salvo and the remaining possible ship instances are then OVERLAYed as before.

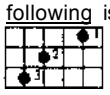
Lastly, since blocks of the salvo itself will be included in the heuristic diagrams generated from damage reports of one or more hits, these blocks need to be removed from the final diagrammatic heuristic as they are not available for future salvos. This is accomplished by ANDing the negation of the current board with the heuristic diagram developed so far. Thus the entire process of developing a heuristic diagram for the constrained game of Battleship can be formalized as:

```
¬ CurrentBoard
∧
ACCUMULATE(
    Ø,
    ShipPositions :=
      FILTER(λ(x) Hits=
                  CARDINALITY(
                    FILTER(NONNULL,
                           MAP(λ(y) (y ∧ CurrentSalvo ∧ x),
                               SingleBlocks)),
              ShipPositions),
    )
```

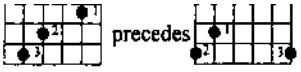
In summary, this heuristic diagram is computed from information about where ships cannot be from previous salvos (in the previous heuristic diagram) and the hit information from the current salvo. It provides guidance for the aggressor's next shots and provides information for the next heuristic diagram in the sequence.

4.2 Guitar Chord Notation

Guitar chord notation attempts to denote positions of fingers for a given chord by diagrammatically representing strings and frets of a fingerboard along with dots to represent finger positions. Syntactically, vertical lines represent the *strings* of a guitar whereas horizontal lines represent *its frets.* A dot on a string represents where some finger is placed to produce a desired pitch. Semantically, a *fingering* is a

specification of exactly which of four fingers to use to realize the dots of the diagram. For example, given that numbers 1 through 4 represent the index finger to the little finger, the following is a chord diagram complete with a fingering: 

Chord diagrams are superior to standard musical notation for inferring fingering information since the fingerboard positioning of the chord is explicitly shown on the diagram but must be inferred from standard musical notation. Even so, semantic ambiguity arises in guitar chord diagrams because 1) fingerings are often not specified and 2) there exists a one-to-many mapping between the dots and possible fingerings. A given chord can sometimes be fingered many ways with the preferred way often being context dependent. That is, the preferred fingering of a chord will often depend on one or both of the chords preceding and following it in the diagram sequence. For example, when

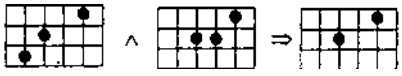


there is no dot in common between them and, therefore, the fingering for the second chord defaults to its least demanding state as shown. If, however,

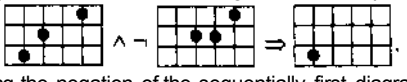
there is a dot in common and the fingering for the second chord attempts to conserve finger movement by leaving the fourth finger in place as shown.

The atomic element in this domain is represented as dots. Each dot can either be black or white; no intermediate gray values are necessary for this domain. Applications of simple diagrammatic operators to sequences of chord diagrams (C, and C_2) produce useful information.

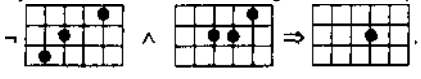
ANDing two such diagrams will yield a new diagram that is comprised of all black dots these diagrams have in common, more formally C_1 A C_2 => *CommonFingers.* For example,



ANDing the sequentially first diagram with the negation of the second will yield a new diagram that is comprised of only those black dots that were *removed* over time, more formally, C_t ^C_2 => *RemovedFingers.* For example,

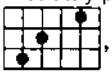
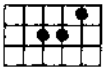


ANDing the negation of the sequentially first diagram with the second will yield a new diagram that is comprised of only those black dots that were *introduced* over time, more formally, - C_1 A C_2 => *IntroducedFingers.* For example,



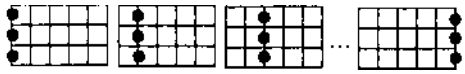
(Note that a background grid of strings and frets must also be merged to the above in order to produce the indicated results.)

Inferring a Chord Fingering

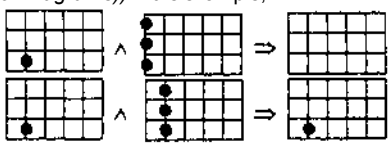
The reasoning goal in this domain is to infer a fingering for a chord in a sequence of chord diagrams given a fingering for the chord immediately preceding it. For example, given a fingering for  , a fingering for  can be inferred via diagrams generated by the postulated operations in concert with two simple, domain specific rules: 1) whenever possible keep fingers in the same position, and 2) when a new finger is needed, use next numerically available finger.

Since there can only be one significant finger on any given string, we can represent a fingering for a given chord by *a fingering vector, [s_6, s_5> S_4, s_3, s_2* s_1,* where each *s,* is a finger number 0 through 4 signifying which (if any) finger is to be placed on the string i. (The strings on a guitar are numbered from lowest pitch to highest pitch as 6 through 1.) The fingering that will be used for the first chord in this example will be represented as [0,3,2,0,1,0]. A list of available fingers can be represented by an *available finger set* that contains the numbers of all fingers not currently in use by a chord. This can be generated for any given chord by inspecting its fingering vector. Thus, the fingers not in use by the first chord in the example is represented as {4}.

The first step to inferring a fingering for the second chord is to update the available finger set with newly available fingers. These can be found by inspecting *RemovedFingers.* Any finger that was used to realize a dot that was removed from the first chord is now available. To accomplish this inspection, six *InspectionDiagrams , id,,* are defined, each associated with one string in the diagram:



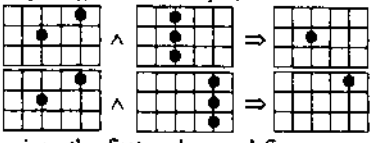
When each of these are individually ANDed with a given diagram, a background grid devoid of dots *(NullChord)* will result whenever there is no dot on the string associated with the inspection diagram in the diagram under inspection. Further, whenever there does exist a dot on the currently inspected string, such ANDing will infer a diagram comprised of the background grid with that dot in place. Formally stated: FILTER (NONNULL, MAP(*l(x) {RemovedFingers* A X), *InspectionDiagrams))* In the example,



Since the only inference that produces anything other than the background grid is the one involving the fifth string's

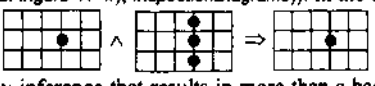
inspection diagram, the finger that was used on s_5 in the fingering vector (namely, finger number three) is now available. The available finger set is then updated to {3,4}.

A similar approach is taken to find which fingers (if any) should remain in the positions they occupied in the first chord diagram. Each inspection diagram is ANDed, in turn, to *CommonFingers*. The inferences that do not produce an empty background grid are exactly those made with inspection diagrams associated with the strings that should maintain the same fingering. These fingers are then transferred to the new fingering under development. Formally stated: FILTER (NONNULL, MAP($\lambda(x)$ (*CommonFingers* $\wedge$ x), *InspectionDiagrams*)). In the example,



therefore, since the first and second fingers were used on the second and fourth strings, respectively, the partial fingering for the next chord that keeps these fingers the same is [0,0,2,0,1,0] given a fingering vector that is initialized to all zeros.

Next, *IntroducedFingers* is used to find which strings require new fingers (if any). Again, the inspection diagrams are used to find which diagrams have dots in this result. Those that do need to have fingers placed on their associated strings. Formally stated: FILTER (NONNULL, MAP($\lambda(x)$ (*IntroducedFingers* $\wedge$ x), *InspectionDiagrams*)). In the example,



is the only inference that results in more than a background grid, so a new finger is needed on the third string. This finger is retrieved from the available finger set via a *minimum* function that returns the smallest finger number in the set, in the example, 3. This finger number is then placed in the vector at the place corresponding to the string that needs a finger, s_3. The final fingering vector for the second chord diagram in the example is [0,0,2,3,1,0].

This example shows how diagrammatic representations can be combined with other knowledge representations to infer new knowledge. The indices associated with each inspection diagram serve as the intersection between the diagrammatic inferences and the symbolic inferences. It is our expectation that diagrammatic representations will prove to be useful in many instances of such multimodal reasoning.

4.3 Cartograms

Another application of the postulated diagrammatic operators is in the domain of cartograms, in particular, *chloropleths* — maps that present data as areas of different colors or gray-level intensities. When such maps depict different data for the same geographic location, they are related in precisely the manner needed to perform diagrammatic inference.

The *TemperatureMap* and *PrecipitationMap* chloropleths depicted in Figure 5 comprise a diagram suite in that they are two different aspects of a geographic location that occur simultaneously. *TemperatureMap* is a map in which temperatures are denoted by different gray scale intensities— darker signifying higher temperatures and lighter signifying lower temperatures. *PrecipitationMap* is a map that denotes by some gray value where it is currently precipitating.

Inferring a New Chloropleth

Figure 5 details how the proposed diagrammatic operators can be used to infer a new chloropleth that represents where it is likely to be snowing within the given geographic location by finding the intersection of the area where it is thirty degrees or below and the area where it is precipitating. A cartogram covered with the gray value that represents the thirty degree range (denoted *30DegreeIntensityMap)* is PEELed from *TemperatureMap,* forcing all gray values representing the thirty degree range and below to WHITE while only lightening other grays. The result of this operation is then negated forcing all non-WHITE areas to WHITE and the thirty degree and below area to BLACK. Finally, this is ANDed with *PrecipitationMap* producing a new chloropleth that represents the area within the given geographic location in which it is in the thirties or below and is precipitating— by definition the most likely area for snow. Formally, this process can be represented as:

$$\neg(\textit{TemperatureMap} - \textit{30DegreeIntensityMap}) \wedge \textit{PrecipitationMap}.$$

There are many similar inferences that can be made with such maps. Further, other questions could be asked of a system that dealt with this type of data. For instance, given a set of inspection diagrams representing the states, questions such as "In which states is it precipitating?" , "How many states currently have temperatures below thirty degrees?", or "Is it likely to be snowing in Connecticut?" can be easily answered diagrammatically. We envision a geographic database of diagrammatic data with queries presented

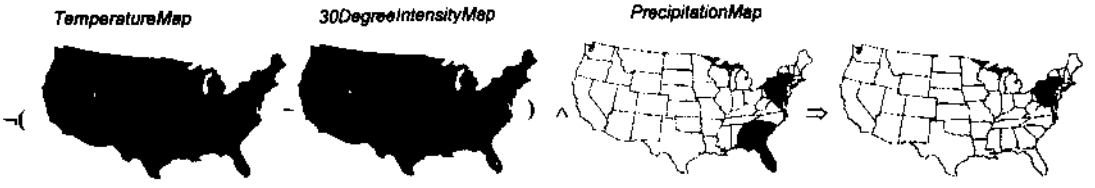


Figure 5: Inferring where it is most likely to be snowing

diagrammatically and resolved by diagrammatic inferencing as a reasonable goal of our research.

5 Related Work

As previously stated, little work has been done with diagram sequences and suites per se. One notable exception is the work done by Bieger and Glock [1985; 1986]. Beside attempting a taxonomy of categories of information presented in what they term "picture-text instructions", they performed rigorous experimentation with actual subjects and monitored their use of such instructions to the end of identifying the most critical categories. The direction of their work is not towards automating diagrammatic reasoning but towards understanding human use of such information as is work by Willows and Houghton [1987].

Furnas [1992] postulates a logic that deals with diagrams via BITPICT rule mappings that can be used to transform one diagram into another and, therefore, allows reasoning from diagrams to diagrams. Interesting as this reasoning is, these explicit rule mappings can be subsumed by the operators and functions currently proposed. Further, Furnas' work does not attempt to reason *about* diagrams in sequences but, rather, its crux is the *generation* of sequences of diagrams to accomplish some reasoning goal pertaining to a single diagram.

Lastly, Chapman [1991] posits a number of primitive *visual operators* that are used as building blocks of various task-specific *visual routines.* These operators process information about single frames of a video game; no reasoning about sequences of frames takes place. Interestingly, though, two main operations of his work have analogs in the inter-diagrammatic operators we have postulated. *Visual attention,* focusing on a subset of a scene, can be seen as analogous to the PEELing and negating of a cloropleth to concentrate on one particular gray-level intensity. *Visual search,* finding particular elements in a scene, is analogous to our use of inspection diagrams.

6 Conclusion

We have postulated a number of diagrammatic operators and functions and have shown how they can be useful in reasoning with sequences and suites of related diagrams. In particular, we have used them in the domains of game playing, musical notation, and cartograms. These operators and functions provide a clear and concise means of describing diagrammatic manipulations in an important subset of diagram groupings, namely, those in which objects are identifiable by their positions within diagrams. This is a first step in an attempt to endow a system with full diagrammatic reasoning capabilities that we believe will prove extensible to other, less constrained groupings of diagrams.

In general, we believe that a diagrammatic reasoning approach that deals with diagrammatic representations directly will often prove beneficial because it will avoid the difficult problems of 1) generating exhaustive textual or other representations that specify all objects and their relationships and 2) focussing attention on only pertinent objects and relationships. Further, when viewed at the pixel level, the postulated diagrammatic reasoning operators are closely related to raster graphics operators and, therefore, it is possible that hardware optimized for such graphics operators could efficiently perform diagrammatic operations.

References

[Anderson, 1994] Anderson, M., "Reasoning with Diagram Sequences", *Proceedings of the Conference on Information-Oriented Approaches to Logic, Language and Computation (Fourth Conference on Situation Theory and its Applications),* 1994

[Anderson and McCartney, 1995] Anderson, M. *and McCartney, R., "Developing a Heuristic via Diagrammatic Reasoning", *Proceedings of the 1995 ACM Symposium on Applied Computing,* 227-231, 1995

[Bieger and Glock, 1985] Bieger, G. and Glock, M., "The Information Content of Picture-Text Instructions", *The Journal of Experimental Education,* 53(2), 68-76, 1985

[Bieger and Glock, 1986] Bieger, G. and Glock, M., "Comprehending Spatial and Contextual Information in Picture-Text Instructions", *The Journal of Experimental Education,* 54(4), 181-188, 1986

[Chandrasekaran *et al,* 1993] Chandrasekaran, B., Narayanan, N. and Iwasaki, Y., "Reasoning with Diagrammatic Representations", *A] Magazine,* 14(2), 1993

[Chapman, 1991] Chapman, D„ *Vision, Instruction and Action,* MIT Press, 1991

[Furnas, 1992] Furnas, G., "Reasoning with Diagrams Only", in [Narayanan, 1992]

[Larkin and Simon, 1987] Larkin, J. and Simon, H., "Why a Diagram is (Sometimes) Worth Ten Thousand Words", Cognitive Science, 11, 65-99, 1987

[Narayanan, 1992] Narayanan, N., editor, *Working Notes of AAAI Spring Symposium on Reasoning with Diagrammatic Representations,* 1992

[Narayanan, 1993] Narayanan, N., editor, "Taking Issue/Forum: The Imagery Debate Revisited", *Computational Intelligence,* 9(4), 1993

[Willows and Houghton, 1987] Willows, D. and Houghton, H., *The Psychology of Illustration,* Springer-Verlag, 1987