Proteus: Visual Analogy in Problem Solving

Jim Davies [a] Ashok K. Goel [b] Patrick W. Yaner [b]

[a] *School of Computing*
Queen's University; Kingston, ON, K7L 3N6 Canada

[b] *College of Computing*
Georgia Institute of Technology Atlanta, GA 30332 USA

Abstract

This work examines the hypothesis that visual knowledge alone is sufficient for analogical transfer of problem-solving procedures. It develops a computational theory of visual analogy in problem solving which has been implemented in a computer program called Proteus. Proteus provides two main things. Firstly, it provides a content account for visual analogy in problem solving, along with a corresponding vocabulary and data structures for representing the knowledge content. Secondly, Proteus provides a process account for visual analogy in problem solving, along with corresponding methods and algorithms. Proteus addresses all major subtasks of analogy. It also identifies a new subtask in the task stucture of analogical problem solving: dynamic generation of new mappings between the intermediate knowledge states in the source and the target cases when a step in the transferred procedure creates a new object. Finally, by examining the limitations of use of visual knowledge alone, Proteus helps identify the functional roles of (non-visual) causal knowledge in analogical problem solving.

Key words: analogy, visual reasoning, visual knowledge, problem-solving, case-based reasoning, diagrammatic reasoning, diagrams

1 Introduction

Visual analogy is a topic of longstanding and growing interest in AI. Evans' early ANALOGY program solved multiple choice geometric analogy problems of the kind found on many intelligence tests [9]. Figure 1 illustrates this kind

Email addresses: jim@jimdavies.org (Jim Davies), goel@cc.gatech.edu (Ashok K. Goel), yaner@cc.gatech.edu (Patrick W. Yaner).

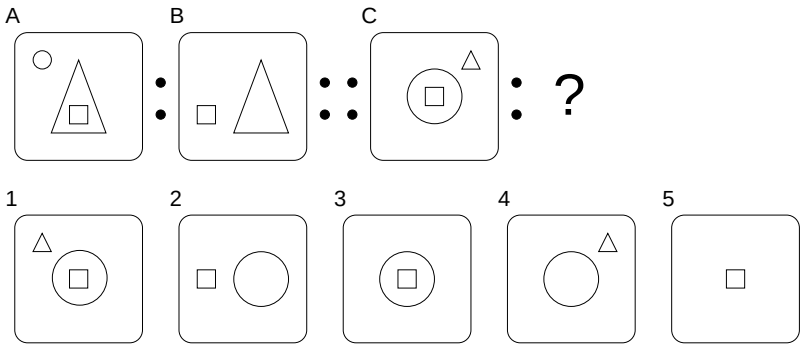


Fig. 1. Evans' ANALOGY program solved problems in geometric analogy. Given A, B, and C, the problem is to determine which one of the choices 1–5 has a relationship to C that is most similar to the relationship B has to A.

of A:B::C:? problem. To address this problem, ANALOGY first found the relational differences between A and B, then found the relational differences between C and each of the available choices, and finally selected the drawing whose relational differences with C were most similar to the relational differences between A and B. The more recent Letter Spirit system [35] takes a stylized seed letter as input (e.g., f but with the crossbar suppressed) and outputs an entire font, from a to z, in the same style as the seed. Letter Spirit addresses this problem by first finding the base letter most similar to the seed letter and thus determining what letter is presented as the seed (e.g., f), then determining the relational differences base letter and the seed letter (e.g., crossbar in f is suppressed), and finally drawing similar components of other letters in the same way as the seed letter. Thus, when Letter Spirit makes a lower-case t, by analogy to the seed letter, it suppresses the crossbar. Note the mappings between letter components are already in the system: the vertical bar part of the letter d maps to the vertical bar in the letter b, for example, and similarly, the crossbar of t maps to the crossbar of f.

Neither ANALOGY nor Letter Spirit, however, engage in analogical *problem solving*, a central issue in AI. This poses the primary question for this research: is visual knowledge alone sufficient for analogical problem solving? Problem solving may be characterized as generation of *procedures* that contain two or more steps. Typically, the procedures are strongly-ordered in that certain steps *must* precede others. It follows that analogical problem solving involves the transfer of a procedure from a source (or base) case to a target problem. We start with the initial hypothesis that visual knowledge alone is sufficient for analogical transfer of strongly-ordered procedures. This leads to the first goal of our work: to develop a computational theory of visual analogy in problem solving.

As ANALOGY and Letter Spirit illustrate, analogies in general address new problems by finding relational similarities and differences between a target problem and a source case and transferring some knowledge (or action) from

the source to the target. In general, analogy involves several subtasks including retrieving from memory the source case most similar to the target problem, mapping (or aligning) the the elements of target and the source, transferring knowledge from the source to the target, evaluating what was transferred in the context of the target, and storing the target in memory. These tasks may themselves involve additional subtasks. For example, the retrieval task may be decomposed into the subtasks of reminding and selection, and the transfer task may involve the subtask of adaptation. ANALOGY addresses only the mapping and transfer subtasks of analogy. In contrast, Letter Spirit addresses only the retrieval, transfer and evaluation subtasks (since the mappings between different letters are already stored in the system). A second goal of our work is to build a unified theory of visual analogy that not only addresses all major subtasks of analogy, but also uses a uniform knowledge representation for all the subtasks.

Also as illustrated by ANALOGY and Letter Spirit, visual analogy refers to analogy based only on the *appearance* of a situation. e.g., the shape of the letter f and the spatial relationship between its components. Causal and functional knowledge is either not present or is (at most) implicit. Thus, in visual analogy, knowledge states in source cases and target problems are characterized by shapes of objects (e.g., a line, a semi-circle, etc.), and spatial relations among the objects or their components (e.g., above, left-of, contained-in, etc.). Both ANALOGY and Letter Spirit describe a content account of shapes and spatial relations in their respective domains. Our work similarly describes a content account of shapes and spatial relations for visual analogy in problem solving, and provides a vocabulary and data structures for representing the content.

In addition, both ANALOGY and Letter Spirit provide a process account for their respective tasks. Their process account is articulated in terms of their task decompositions, methods for accomplishing specific tasks in the task structure, and algorithms corresponding to the methods. Our work similarly provides a process account for visual analogy in problem solving in terms of task structures, methods and algorithms. A significant finding of our work is that if and when a step in the problem-solving procedure being tranferred from a source case to a target problem *creates new objects*, then the analogical process needs to *dynamically generate a new mapping* between the corresponding intermediate states in the source case and the target problem.

The Proteus system is an implementation of our computational theory of visual analogy in problem solving. Proteus addresses all major subtasks of analogical problem solving. However, since visual representations do not capture (non-visual) causal and functional knowledge, the mapping task, as we will describe in detail below, generates multiple initial mappings between a retrieved source case and the target problem. Further, since the evaluation subtask appears to require causal and functional knowledge, and since causality is (at

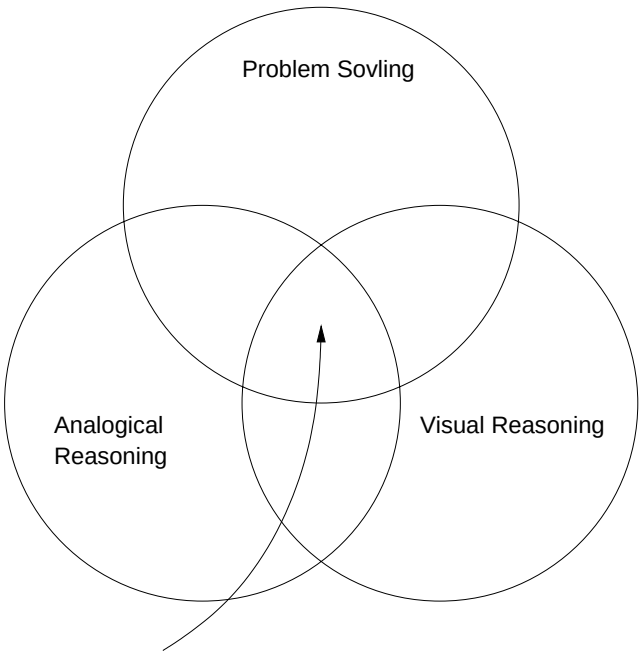


Fig. 2. At the task level, Proteus is a theory of problem solving. At the level of methods, it uses analogical reasoning to solve problems. In terms of types of knowledge, it uses only visual knowledge for making analogies.

most) only implicit in visual representations, Proteus does not automate the evaluation task with visual reasoning. By examining the limitations of use of visual knowledge alone, Proteus helps identify the necessary functional roles of causal knowledge in analogical problem solving.

2 Proteus

Knowledge states in Proteus are represented as 2-D line drawings generated by vector graphics programs such as Xfig and Jfig. We will call these states `s-images` (an abbreviation of 'symbolic images,' as they contain imagistic information represented symbolically). Proteus takes as input an `s-image` representing an unsolved target problem, and outputs a procedure to solve the target problem, where the output procedure is a sequence of `s-images` and transfromations between them. Figure 3 illustrates what Proteus generates once a source analog is selected. Given an `s-image` representing the target problem (at the bottom-left of the figure), Proteus generates a procedure for solving the problem in the form of a sequence of `s-images` (shown in the gray area in figure).

Proteus contains a long-term memory of source cases, where each source case contains a problem-solving procedure (as illustrated in Figure 3), and is indexed by an `s-image` repersenting the initial knowledge state in the procedure.

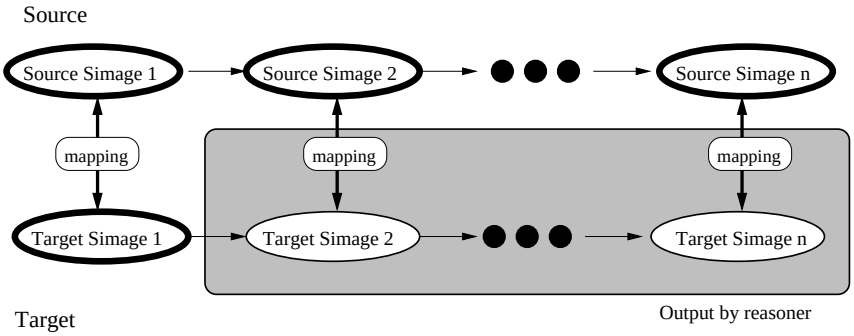


Fig. 3. This figure illustrates the input and output to Proteus' transfer in the abstract. The input is an `s-image` representing the target problem (shown at the bottom left). The output is a procedure represented as a sequence of `s-images` (shown in the gray area at the bottom-right of the figure). Proteus generates the procedure by transferring a procedure from a source case (shown at the top of the figure). In the process, it retrieves source cases, generates mappings between the target problem and the source case as indicated in the figure, and so on.

Proteus solves an input target problem by executing the five major subtasks of analogy. The first task is retrieval, in which the target problem, represented by a single `s-image`, is used as a query into the case base. If the retrieval task outputs multiple source cases, then Proteus selects one for further processing. The second subtask is mapping (as indicated in Figure 3), in which the elements (i.e., objects, relations) of `s-image` representing the target problem are matched with corresponding elements in the `s-image` with which the source case is indexed. The output of the mapping task is a series of maps, each linking an element in the source to an element in the target. The third subtask is transfer. Proteus transfers the steps in the procedure in the source to the target problem one step at a time. The transfer task may also involve adaptation of some elements in the source case. The fourth step is evaluation. Proteus presently uses precompiled knowledge for evaluating the solution to the target problem, as described below. If the evaluation fails, then it goes back to the output of the retrieval task, and selects a different source case for processing.

2.1 An Illustrative Example

We will use the classic fortress/tumor problem [8] as the running example throughout this paper. In this problem, a general must overthrow a dictator in a fortress. His army is poised to attack along one of many roads leading to the fortress when the general finds that the roads are mined such that large groups passing will set them off. To solve the problem, the general breaks the army into smaller groups, which take different roads simultaneously, arriving together at the fortress. In the unsolved target problem there is a tumor that must be destroyed with a ray of radiation, but the ray will destroy healthy

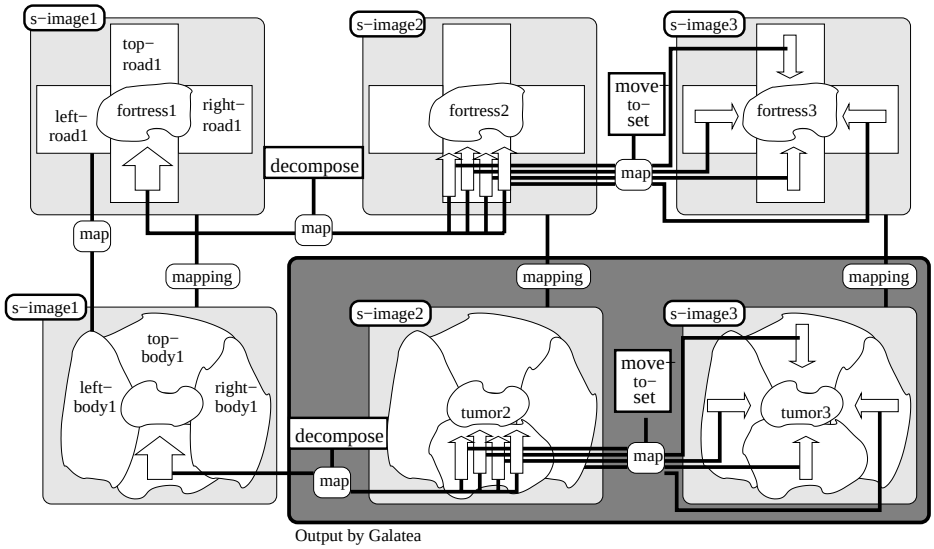


Fig. 4. This figure illustrates the input and output of the transfer stage in Proteus for the fortress/tumor problem. The input is an `s-image` representing the target tumor problem (shown at the bottom left). The output is a procedure represented as a sequence of `s-images` (shown in the gray area at the bottom-right of the figure). Proteus generates the procedure by transferring a procedure from the source fortress case (shown at the top of the figure). In the process, it retrieves source cases, generates mappings between the target problem and the source case as indicated in the figure, and so on.

tissue on the way in, killing the patient. The analogous solution is to have several weaker rays simultaneously converging on the tumor to destroy it. Figure 4 illustrates Proteus' task for the fortress/tumor problem.

2.2 Knowledge Representation

It is important that Proteus use an uniform knowledge representation for all subtasks of visual analogy in problem solving. Drawing in part on the literature on visual reasoning (e.g., [22,1]), and partly by trial and error, we designed a knowledge representation language called *Covlan* (for Cognitive Visual Language). Covlan provides a vocabulary for representing (a) primitive visual elements in an `s-image` (such as a circle, a set, a connection), (b) primitive visual relations `s-image` (such as touching, above, left-of), (c) qualitative variables in `s-images` (such as locations, sizes, thicknesses), (d) primitive transformations that apply to the visual elements and relations, and change the elements, relations and/or the values of variables (such as move, decompose, replicate), and (e) mappings between `s-images`. Of course, Covlan in its present form is incomplete; a different class of problems than the one we have studied may require additional primitives, which may provide additional expressivity and precision.

Primitive Visual Transformations	
Transformation name	**arguments**
add-element	object-type, location (optional)
add-connections	connection/connection-set
decompose	object, number-of-resultants, type
move-to-location	object, new-location
move-to-set	object, object2
put-between	object, object2, object3
replicate	object, number-of-resultants

Table 1
Primitive transformations.

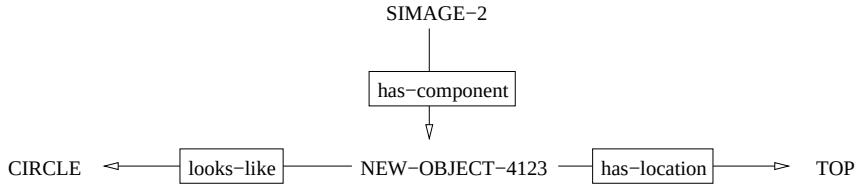


Fig. 5. A graphical representation of the three relationships added to an `s-image` by the `add-component` transformation. `Relations` are boxed. beginning of arrows are the in the

2.2.1 Primitive Transformations

Table 1 shows Covlan's ontology of transformations. The application of the first entry, `add-element`, to an `s-image` adds a new primitive element in the next `s-image` in a procedure. The first argument, `object-type`, must be an instance of one of the `primitive elements` (e.g. `square` or `circle`, described below). It determines what kind of shape appears in the next `s-image`. The second argument specifies the `location` of the shape. Covlan uses qualitative `locations`: `bottom`, `top`, `right`, `left`, or `center`. As Figure 5 illustrates, the application of `add-element` builds a representation for the next `s-image` by adding three propositions to the representation of the current `s-image` (all of Proteus' memory is in propositional form: a relationship connects two concepts with a relation.) (1) a `has-component` relation with the name identifying the new component, 2) the new component's name with a `looks-like` relation to the `object-type`, and 3) the component's name with a `has-location` relation to the `location` input as an argument.

`Add-connections` is a transformation that inserts a set of connections in the next `s-image`. The input is the name of the set of connections in the source. To determine the nature of the connections in the target, Proteus uses substitution for all the symbols in the source and the target to find the analogous

names, so that analogous connections are placed in the next target `s-image`.

`Decompose` takes a primitive element and replaces it in the next `s-image` with some `n` elements. It also correspondingly reduces the thickness for each of those elements.

`Move-to-location` changes the location of a primitive element from one location to another. This means that in the next `s-image`, the old `has-location` relation is removed and a new `has-location` relation is added, relating the element to the input `location`, which can be an absolute location or another element instance (in which the two element instances are co-located.)

`Move-to-set` takes in two sets as input (we will call them *set-a* and *set-b*). The members of *set-a* are moved to the locations of the members of *set-b*. If *set-a* and *set-b* have the same number of element instances, then each element of *set-a* is placed on a distinct element in *set-b*. The element instance matching is arbitrary. If *set-a* has more elements, then multiple members of *set-a* are placed at the locations of each member of *set-b*. The number of element instances in these groups is determined by the number of elements in *set-b* divided by the number of elements in *set- a*. If *set-b* has more elements, then elements of *set-a* are distributed across the locations of the members of *set-b*.

`Put-between` takes two objects that are `touching`, and places some third object in between them. In the new `s-image` 1) the two objects are no longer touching and 2) the third is touching both of them.

`Replicate` takes in an element or set of elements and generates `n` new instances of that element or elements in the next `s-image`. Its behavior is similar to `decompose`, except that it does not change the size or thickness of elements, and can work on sets as well as single element instances.

2.2.2 Primitive Elements

Covlan's ontology of `primitive visual elements` (Table 2) contains: `rectangle`, `circle`, `line`, and `set`. The elements are represented as framess with slots that can hold values. For example, a `rectangle` has a `location`, `size`, `height`, `width`, and `orientation`. All elements have a `location`, which holds a value representing an absolute location on an `s-image` (e.g. `top`, `right`).

The `connection` is a special `element` which we describe under primitive relations. The `set` is another special `element`. A `set` can contain any number of `instances` of `elements`. `Sets` also have an `orientation`, the value of which is one of the primitive `directions` (described below). An `element instance` in the `set` is specified in the representation as the `front`, and another element

Primitive Visual Elements	
Element name	**attributes**
connection	subject, object, angle, distance
rectangle	location, size, height, width, orientation
circle	location, size, height
line	location, length, end-point1, end-point2, thickness
set	location, orientation, front, middle

Table 2
Primitive elements.

is specified as the `middle`. The orientation is defined as an (imaginary) line from the `middle` to the `front` in the `direction` specified in the `orientation`.

Sometimes a *part* of an `element instance` must be referenced. For example, if a line touches the middle of another line, there must be some way to describe the *end* of the first line and the *middle* of the next. To support this, in Covlan different primitive elements have different kinds of `areas` such as *middle* and *end*.

`Lines` have `start` and `end points`, as well as `right-side` and `left-side mid-points`. The `element instance`'s names are related to the symbols naming these areas (e.g. `line1-end-point` with `area-relations`: `has-end-point`, `has-start-point`, `has-rightsidemiddle`, and `has-leftsidemiddle`.)

`Circles`, `squares`, and `rectangles` have `sides`, which are related to element instances with the following relations: `has-side1` (the top), `has-side2` (the right side), `has-side3` (the bottom), and `has-side4` (the left side).

Table 3 shows some of the visual elements and their attribute values for the first `s-image` in the fortress problem.

2.2.3 Qualitative Variables

Qualitative variables are symbols that can take a qualitative value for element attributes or transformation arguments. They can be broken down into the following types: `angles`, `locations` [1] , `sizes`, `thicknesses`, `numbers`, `speeds`, `directions`, and `lengths`.

Fortress Problem Elements		
Visual Object	**attributes**	**value**
Fortress	looks-like: location:	curve center
Bottom-road	looks-like:	line
Right-road	looks-like:	line
Left-road	looks-like:	line
Top-road	looks-like:	line
Soldier-path	looks-like: location: thickness:	line bottom-road thick

Table 3
Primitive elements from fortress problem `s-image` 1.

Qualitative Variables	
angles	perpendicular-angle, right-angle-cw, 45-angle-cw, 45-angle-ccw, right-angle-ccw
locations	bottom, top, right, center, off-bottom off-top, off-right, off-left
sizes	small, medium, large
thicknesses	thin, thick, very-thick
speeds	slow, medium, fast
directions	left, right, up, down
lengths	short, medium, long

Table 4
Classification of Variables

Visual Relations	
Visual Relations	touching, above-below, right-of-left-of, in-front-of-behind, off-s-image
Motion Relations	rotating, not-rotating

Table 5
Primitive visual relations.

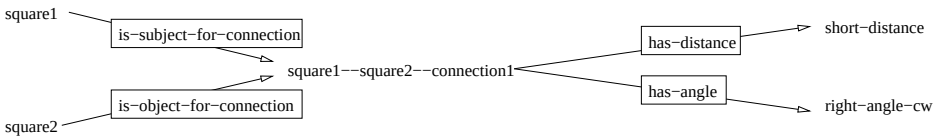


Fig. 6. A graphical representation of the relationships involved with a connection. `Square2` is a short distance to the right of `square1`. `Right-angle-cw` means that the angle is a right angle in the clock-wise direction.

2.2.4 Primitive Relations

The class of `primitive visual relations` (shown in Table 5) describe how certain visual elements relate to each other and the `variables`. `Motion relations` (see Table 5) describe how element instances are moving in an `s-image`. `Rotation` has the arguments `speed` and `direction`.

Many spatial relationships between primitive elements are represented with `connections`. A `connection` is a primitive element with a `name`. `Connections` are frames with two four-slots: `subject`, `object`, `angle` and `distance`, represented with `is-subject-for-connection`, `is-object-for-connection`, `has-angle` and `has-distance`. These relations connect the `connection` name to `distances` and `angles`, which are `qualitative variables`, as illustrated in Figure 6. The `object` of the `connection` is `distance` away from the `subject` in the direction of `angle`.

The `distances` are `touching-distance`, `short-distance` and `long-distance`. The `angles` are `perpendicular-angle` (straight ahead), `right-angle-cw` (a right angle in the clockwise direction, or to the right), `45-angle-cw` (a forty-five degree angle to the right), `45-angle-ccw` (a forty-five degree angle in the counter-clockwise direction, or to the left), and `right-angle-ccw` (a right angle to the left). Figure 7 illustrates the different kinds of `connections` Covlan can represent. `Areas` of `element instances`, as well as `element instances` themselves can be connected.

2.2.5 Mappings between `S-images`

An `s-image` in the source can have an `analogy` between it and its corresponding `s-image` in the target. Each analogy can have any number of analogical `mappings` associated with it (determining which `mapping` is the best is the *mapping problem.*) Each alignment between two `element instances` or `areas` in a given `mapping` is called a `map`. [2]

[1] Relative locations, as opposed to absolute locations, are classified under `primitive visual relations`.
[2] A `map` is called a `match hypothesis` in the SME literature [10].

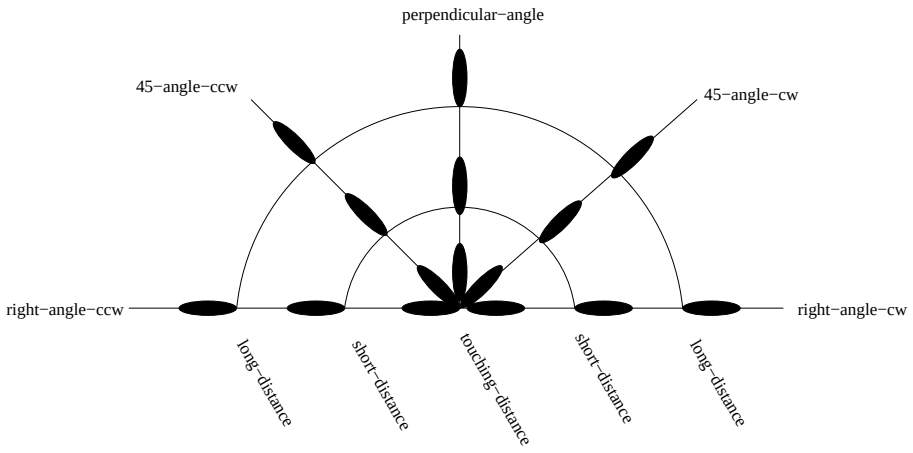


Fig. 7. Each of the fifteen black dots in the Figure represents a qualitative `connection` area, with an `angle` and `direction`.

Similarly `s-images` next to each other in sequences have `transform-connections`. These are necessary so Proteus can track how visual elements in a previous `s-image` change in the next. A difference between `analogies` and `transform-connections` are that there can be multiple analogical `mappings` for an `analogy`, but only one `mapping` for a `transform-connection`. `Transformations` are attached to a `map` between two `element instances` in sequential `s-images`. Thus, if a `rectangle` changes into a `circle`, Proteus knows which `rectangle` in the previous `s-image` turns into which `circle` in the next `s-image`.

Proteus represents the fortress case as a sequence of three `s-images` as illustrated in Figure 4.) The first `s-image` is a representation of the original fortress problem. It had `n` roads, represented as thick lines, radiating out from the fortress, which was a `curve` in the `center` (`curves` are used to represent irregular shapes). Proteus represents the original soldier path as a thick `line` on the bottom road. This `s-image` was connected to the second with a `decompose transformation`, where the arguments were `soldier-path1` for the `object` and `four` for the `number-of-resultants`. The second `s-image` shows the `soldier-path1` decomposed into four thin `lines`, all still on the bottom road. The `lines` are thinner to represent smaller groups.

Proteus represents the start state of the tumor problem as a single `s-image`. The tumor itself is represented as a `curve`. The `ray` of radiation is a thick `line` that passes through the bottom body part.

2.3 Retrieval of Source Cases

Proteus' method for retrieval in visual analogies draws on two ideas from earlier work on case retrieval. Firstly, following MAC/FAC [17], Proteus's retrieval method decomposes the retrieval task into two subtasks: reminding (or initial

recall), and selection. Secondly, following ACME [29], Proteus views case retrieval as a constraint satisfaction problem. Note that while MAC/FAC uses structure-mapping [10] for the selecton task, ACME uses a relaxation procedure based on spreading-activation for the retrieval and mapping task as a whole, though there is a complementary system called ARCS [39] which employs much the same method for retrieval by weakening the structural constraints. In contrast, Proteus uses feature vector matching as the method for the first task of reminding, and constraint satisfaction with backtracking for the second task of selection.

The retrieval task is essentially one of matching objects (variables and constants) in the target and the source under the constraints imposed by the propositions in which they appear. An `s-image` can be viewed as a network of relationships between visual elements, and so we can view the `element instances` of the target `s-image` as *variables*, whose potential values are `element instances` of all the source `s-images` in memory, and the links between `element instances` in the target we can view as constraints that must hold between the variables. Thus, each proposition imposes a constraint between two variables in the `s-image`.

Treating the `element-instances` in the target as variables to be assigned values, the potential values are the `element instances` from the `s-image` descriptions in memory, all of which are considered at once. That is, the method is not performing a separate test on each potential source in memory, but, rather, it is running a search procedure on the entire memory considered collectively. The constraints on the values assigned to the variables (the target `elements`) are precisely those imposed by the subgraph isomorphism problem: if `elements` A and B from the target are to be matched with `elements` X and Y from memory, respectively, then, first, X and Y must be in the same `s-image`; second, all relations that hold between A and B must also hold between X and Y, respectively. If these constraints are met, then A can be matched with X and B can be matched with Y. Here, the constraints are binary (say, A is left of B—a relational constraint). The only exception is the constraint that all values be from the same `s-image`, but this can be inferred from the binary constraints.

2.3.1 Retrieval and Matching Process

The matching process works in three phases: initialization of domains, reduction of domains, and finding the matching, where matching means subgraph isomorphism. The first phase initializes the target domains to sets of values that are involved in the same kinds of relations. The second phase reduces these domains by eliminating values that are not all in the same `s-image`. These two phases reduce the selection of values for each variable. The third

function InitDomains

Input:	1. Target `s-image` and Covlan description 2. The memory: memory
Output:	1. A list of potential source `elements` for each target `element`

Procedure:

```
1: Let Nodes be a list of all the nodes in the target
2: for all w ∈ Nodes do
3:    InitDomain[w] ← {}
4:    Let Terms be a list of all the terms in which w appears
5:    MappedNodes ← none
6:    for all term ∈ Terms do
7:       Let Candidates be a list of all nodes from memory incident on
         a term whose label matches term (either incoming or outgoing
         as appropriate)
8:       if MappedNodes = none then
9:          MappedNodes ← Candidates
10:      else
11:         MappedNodes ← Candidates ∩ MappedNodes
12:   InitDomain[w] ← MappedNodes
13: return InitDomain
```

Table 6
Algorithm for InitDomains, which initializes the domains of the target variables—that is, a list of potential source `element instances` for each target `element instance`.

phase actually computes the isomorphism using constraint satisfaction and backtracking.

The first phase (initialize domains) works by finding `element instances` in memory that "look similar" to the target `elements`: if a target `element` A has, say, three relations whose labels are R, S, and T, then the algorithm builds a list of all `elements` in memory—across all the potential source `s-images`—that have at least three relations with labels R, S, and T. We call this the "signature" of an `element`. The second phase (reduce domains) works by ensuring that the set of `s-images` that are represented in the domain of (list of values for) each variable is the same. This serves to eliminate any value from the domain of any variable that does not come from a `s-image` represented in every other variable's domain.

The first stage applies two heuristics to the `s-images` in memory: (1) prune any individual element (as opposed to entire `s-images`) that does not have the same signature as the corresponding target element, and (2) prune any propositions whose associated `s-images` are not represented in every target

function ReduceDomains

Input:	1. The memory: memory 2. List of all `elements` in the target `s-image` ($Nodes$) 3. Initial target domains (list of potential `elements` from memory for each target `element`)
Output:	1. Reduced target domains

Procedure:

```
1: InitDomain ← InitDomains()
2: Determine the associated s-images for each element of each list
   in InitDomains
3: Let ReferenceList be the intersection across all of these lists
4: for all w ∈ Nodes do
5:    Domain[w] ← {}
6:    CurrentList ← {}
7:    for all i ∈ InitDomain[w] do
8:       if the document id of i is in ReferenceList then
9:          Add i to CurrentList
10:   Domain[w] ← CurrentList
11: return Domain
```

Table 7
Algorithm for ReduceDomains, which eliminates from the domains any `element instances` from `s-images` not represented by at least one `element instance` in every domain. Each variable domain is a list of elements, and each element is labelled by the `s-image` that contains it (in line 2, above).

element's domain. The latter enforces subgraph isomorphism. It is important to note that these are both logically implied by the similarity metric that the last phase, described below, implements. It would be an interesting experiment to look at other heuristics that prune out mappings that might have otherwise been returned by the last phase. These two algorithms appear in tables 6 and 7.

The last phase of the retrieval process (find matchings) is the most important step. The basic procedure is one that generates matchings, checking them for consistency as it goes, and backtracking when necessary. The test, here, is actual subgraph isomorphism: if A is related to B in the target, then the relations (links, edges) between any `element` that A maps to and and any `element` that B maps to must include at least those that held between A and B. This algorithm returns all valid mappings. The idea is that the first two phases have restricted the set of possible mappings to search through so that there are not nearly as many, now, as there would have been if a depth-first search without heuristics had been done. This algorithm appears in table 8, with a test performed at each attempt to expand the current mapping given

function FindMatchings

Input:	1. Target `s-image` 2. List of all `elements` in the target `s-image` ($Nodes$) 3. Reduced target domains ($Domains$) 4. The memory
Output:	1. List of all mappings from source to target

Procedure:

1: $Domain \leftarrow$ ReduceDomains()
2: $n \leftarrow$ Length($Nodes$)
3: Let $Mappings$ be nil ($Mappings$ will be a list of lists, each one an complete mapping)
4: $Open \leftarrow \{(\text{nil}, \text{nil})\}$ ($Open$ is a stack of all current partial mappings)
5: **while** $Open \neq \{\}$ **do**
6: $(w, current) \leftarrow$ Pop($Open$)
7: w is the current target element from $Nodes$, or nil of one hasn't been selected yet, and $current$ is the current partial mapping
8: $w \leftarrow$ GetNextElement($Nodes$,w)
9: **for all** $j \in Domain[w]$ **do**
10: **if** Consistent(w,j, $current$) **then**
11: $new \leftarrow$ Append($current$, $w = j$)
12: **if** $w = n$ **then**
13: Push(new,$Mappings$)
14: **else**
15: Push((w, new), $Open$)
16: Each item (list) in $Mappings$ now corresponds to a matching from the target to some document in memory.
17: **return** $Mappings$

Table 8
The algorithm for FindMatchings, which returns mappings for all source `s-images` in memory for which the target `s-image` is a subgraph. Note that, above, $\{(nil, nil)\}$, the initial value of $Open$, is not the empty set $\{\}$; it has one element, an ordered pair of nils. The first item in the pair is the current target element (nil at the start, indicating one hasn't been chosen yet), and the second is the current mapping (nil at the start, indicating an empty mapping, which is added to as the search progresses)

in table 9.

2.3.2 Mapping Between Source and Target Cases

For the sake of retrieval and mapping, Proteus first makes an index of all the chunks in memory (a chunk is simply one "unit" of the representation, such as

function Consistent

Input:	1. A potential source `element` j (candidate map) 2. The current (partial) mapping 3. Target `s-image`
Output:	1. True or false

Procedure:

1: **if** *current* = nil **then**
2: **return** True
3: **for all** $i \in \{1, \ldots, w-1\}$ **do**
4: **if** Not all relations between target `elements` i and w can be found among the relations between *current*[i] and j **then**
5: **return** False

Table 9
Algorithm for Consistent, which attempts to determine if the proposed `map` is consistent with the current partial `mapping`.

a relation between two specific `element instances`, or an attribute-value pair together with the associated `element`) in a discrimination tree. This is prior to the calling of InitDomains, above (table 6), so that every access to memory in the above algorithms (InitDomains, ReduceDomains, and FindMatchings, tables 6, 7, and 8). Since only initial problem frames should be retrieved, Proteus looks for any `s-image` which is the first of a finished problem-solution sequence, and only indexes those chunks from each such `s-image` in memory. This prevents, for example, the `tumor-problem s-image` from mapping to the `fortress-solution s-image`, which would be useless: we want it to map to the `fortress-problem s-image`, the first one in the sequence, rather than the last one.

Note that the visual representatins in Proteus capture only structural constraints, e.g, shapes of visual elements, and spatial relations among the elements and between the components of an element. The representations do not explicitly express causal and functional knowledge. Within this context, Proteus' method for mapping between a source case and the target problem employs two key constraints. First, members of sets are not mapped, since they are not employed in transfer (the notion of a set was introduced precisely to facilitate n-to-m mappings). Thus, any `element instance` on the left-hand side of an `in-set relation` is pruned from the input to the mapping task (that is, when the index is getting built). Second, attribute values (e.g. `has-location center`) and qualitative variables (e.g. `has-size small`) are not mapped, only `element instances`, and so they are pruned from the input as well. The alternative to this would be to allow them to be mapped, but then to prune those `maps` from any mappings returned; however, attribute

function RetrieveSourceMappings

Input:	1. A target `s-image` 2. memory
Output:	1. A list of `mappings` from all matching source `s-images` to the target

Procedure:

1: $index \leftarrow$ empty dtree
2: $problemFrames \leftarrow$ all source `s-images` that are the first `s-image` of a solved problem
3: $sourceRels \leftarrow$ all chunks in $problemFrames$ that do not involve `literals` and that do not involve members of sets
4: **for all** $c \in$ all chunks in $problemFrames$ **do**
5: Index(c,$index$)
6: $targetRels \leftarrow$ target `s-image` chunks that do not involve `literals` or members of sets length($sNodes$))
7: $targetNodes \leftarrow$ target `element instances` from $targetRels$
8: $mappings \leftarrow$ findMatchings($sourceRels$, $targetNodes$, $targetRels$, $domains$)
9: **return** $mappings$

Table 10
Algorithm for RetrieveSourceMappings, which searches memory for `s-images` of the problem frames of solved problems for any that match the given target `s-image`, returning a list of mappings.

values may or may not match between source and target, and so they may violate the constraints of subgraph isomorphism, which were introduced as constraints between `elements`, not between attribute values. It is thus more useful to prune them out at the start.

The algorithm in shown in table 10. Note that this mapping algorithm generates multiple mappings instead of a single mapping between a single source and target: it returns *all* mappings from *all* sources that satisfy the given constraints. This is because the mapping algorithm uses only structural constraints present in the visual representation of the source and target `s-images`. It does not also use semantic and pragmatic constraints, such as the goal of the problem solving in the source and target problems, because causal and functional knowledge is not explicitly represented in the visual representation. Therefore, Proteus arbitrarily selects among the multiple mappings, transfers the problem-solving procedure to the target, and evaluates the transferred solution. If the evaluation fails, then Proteus returns to the output of the mapping task, selects a different mapping and repeats the process, until a mapping succeeds or it runs out of mappings to try.

In the fortress/tumor problem, the "correct" mapping maps the set of roads to the set of body parts, the fortress to the tumor, and the army to the ray. When Proteus addresses this problem, the heuristics mentioned above prune out the sets of roads and body parts, as well as the shapes and sizes and positions of all the `element instances`. Thus, the only factors left to influence the mappings were the the `element-instances` themselves. The `element-instances` are `Fortress`, `Tumor`, `Soldier-Path`, `Ray`, `Set1` (the set of roads around the fortress), and `Set2` (the set of body parts surrounding the tumor). Proteus produced only one mapping:

```
(Fortress maps-to Tumor)
(Soldier-Path maps-to Ray)
(Set1 maps-to Set2)
```

In this case, the relationship between the soldier-path and the fortress (namely, that the soldier-path terminates at the fortress) mapped onto the analogous relationship between the ray and the tumor (that the ray terminates at the tumor). This determined the mapping exactly, and no other mappings were even possible under this constraint.

On the other hand, for some of the other examples, a dozen or more mappings were returned. In particular, we discovered an interesting conflict between the needs of retrieval and mapping, and the needs of transfer: for transfer, if only some of the source elements map onto only some of the target elements, transfer can still be possible if the knowledge being transferred does not conflict with anything else in the target. However, for retrieval and mapping, it is straightforward to find all sources in memory for which the given target is a subgraph (viewing the target representation as a graph), but it is not as straightforward to find all sources for which some part of the target may map to some part of the source. Some of the examples involved mapping of the source onto a target which involved other elements in other relationships, and so no mapping could be found using these methods, and the retrieval failed.

2.4 Transfer of the Problem-Solving Procedure

The transfer task takes as input a target problem, represented as a single `s-image`, an source case from memory, represented as a series of `s-images` connected with `transformations`, and sets of possible `mappings` between the target and the first `s-image` of the retrieved source. Proteus transfers solutions from a source to the target and the evaluation step checks the quality of the transferred solutions. Transfer stops when a satisfactory solution is found.

Proteus adapts and transfers each `transformation` in the source problem to the target. The `transformations` are transferred literally and the arguments

of those transformations can be adapted. For example, the transformation decompose is used to turn a primitive element instance into some arbitrary number of resultants, taken as an argument. An argument of a transformation can be an instance of one of three cases. Firstly, the argument can be a literal, like the number four or the location bottom. Literals are transferred unchanged to the target.

Secondly, the argument could be a element instance member of the source s-image. In this case, the transfer procedure operates on the analogous element in the target s-image. For example, in the first transformation in the fortress story, the decomposed source soldier path gets adapted to the ray in the target tumor problem. Thirdly, the argument can be a function.

2.4.1 Transfer Method

We first describe Proteus's transfer method informally, with reference to Figure 4.

(1) **Identify the first s-images of the target and source cases.** These are the current source and target s-images.
(2) **Identify the transformations and their associated arguments in the current s-image of the source case.** This step finds out how the source case gets from its current s-image to the next s-image. In the fortress/tumor example, the transformation is decompose, with four as the number-of-resultants argument (not shown).
(3) **Identify the objects of the transformations.** The object of the transformation is what object, if any, the transformation acts upon. For the decompose transformation, the object is the soldier-path1 (the thick arrow in the top left s-image in Figure 4.)
(4) **Identify the corresponding objects in the target problem.** Ray1 (the thick arrow in the bottom left s-image) is the corresponding component of the source case's soldier-path1, as specified by the mapping between the current source and target s-images (not shown). A single object can be mapped to any number of other objects[3]. If the object in question is mapped to more than one other object in the target, then the same transformation is applied to all of them in the next step.
(5) **Apply the transformation with the arguments to the target problem component.** A new s-image is generated for the target problem (bottom middle) to record the effects of the transformation. The decompose transformation is applied to the ray1, with the argument four. The result can be seen in the bottom middle s-image in Figure 4. The new rays are created for this s-image. Adaptation of the arguments

[3] Though Proteus's mapping generator will not do this, it is possible for mappings associated with transform-connections.

can happen in three ways, as described above: If the argument is an element of the source `s-image`, then its analog is found. If the argument is a `function`, then the `function` is run (note that the `function` itself may have arguments which follow the same adaptation rules as transformation arguments). Otherwise the arguments are transferred literally.

(6) **Map the original objects in the target to the new objects in the target.** A `transform-connection` and `mapping` are created between the target problem `s-image` and the new `s-image` (not shown). `Maps` are created between the corresponding objects. In this example it would mean a `map` between `ray1` in the left bottom `s-image` and the four rays in the second bottom `s-image`. A `map` is also created between the `ray1` to the set of thinner rays. A `mapping` from the correspondences of the first `s-image` enables Proteus to automatically generate updated `mappings` for the subsequent `s-images`.

(7) **Map the new objects of the target case to the corresponding objects in the source case.** Here the rays of the second target `s-image` are mapped to soldier paths in the second source `s-image`. This step is necessary for the later iterations (i.e. going on to another `transformation` and `s-image`). Otherwise the reasoner would have no way of knowing on which parts of the target `s-image` the later transformations would operate.

(8) **Check to see if there are any more source s-images.** If there are not, exit, and the solution is transferred. If there are further `s-images` in the source case, set the current `s-image` equal to the next `s-image` and go to step 1.

Figure 8 shows the third `s-image` in the squence generated for the tumor problem by the above method.

2.4.2 Dynamic Generation of Intermediate Mappings

Step 7 in Proteus' transfer method described in the previous subsection generates a new mapping between the newly generated intermediate knowledge state in the target problem and the corresponding intermediate state in the source case. This is because the `decomposet transformation` preceding this state may create new visual elements in the state. Since this need for dynamic generation of mappings is a new finding of our work, in this subsection we discuss it in some detail.

Consider a hypothetical problem in which two people need the same resource, but only one has access to it. This might be represented visually as illustrated in Figure 9, where the two circles represent people and the triangle represents the one resource. The triangle's proximity to one of the circles might represent which person has possession of the resource.

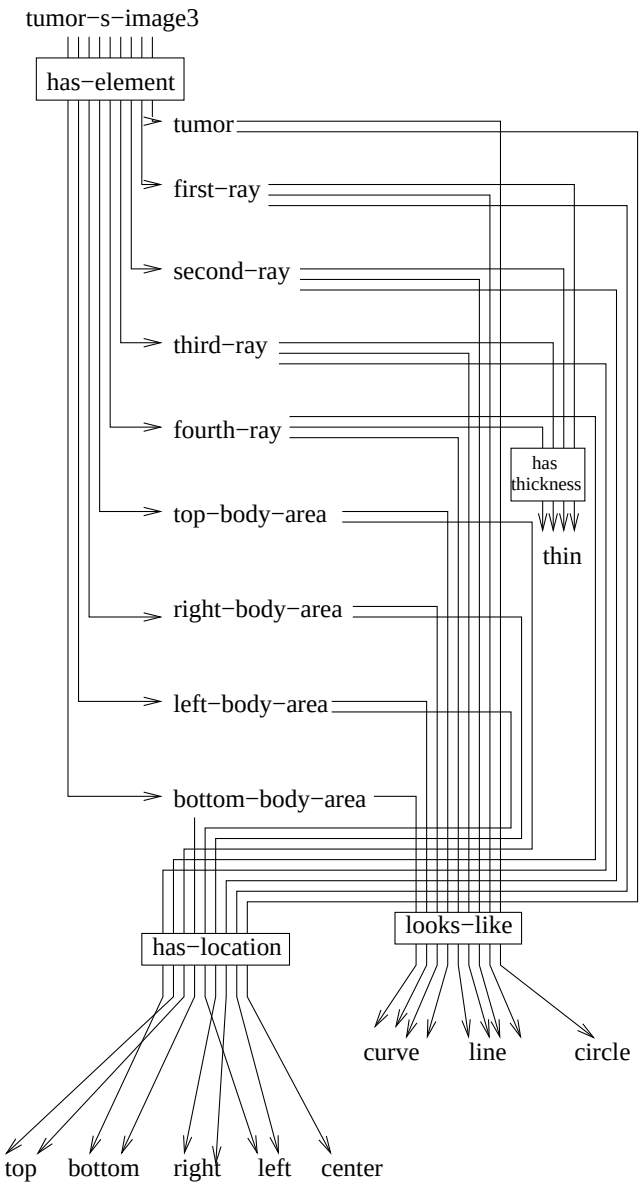
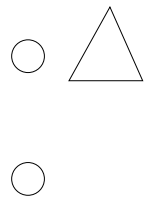


Fig. 8. This Figure shows part of the third `s-image` in procedure for the tumor problem. Each `relationship` is represented as an arrow. The start and ends of the arrows are the ideas connected by the relation in the proposition. The boxed text in the middle of the arrow is the `Relation.` Each string of unboxed text is an `element`, `element instance`, or a `miscellaneous slot value`.

Figure 10 illustrates a problem-solving state in the transfer of a procedure for distributing a resource among people from a hypothetical source case to the target problem. The sequence of `s-images` at the top of Figure 10 illustrates the procedure for distributon of some other resource, depicted as a sqaure, in the source case: the resource is first decomposed into smaller shares, depicted by smaller squares, and then moved to the vicinity of the people. The target problem of Figure 9 is shown in the bottom left of Figure 10. The dotted curves at on the left side in Figure 10 show the initial mappings between the



Initial Target Knowledge State

Fig. 9. A hypothetical target problem. The traingle repersents some resource. Circles represent two people both of whom a share of the resource. The distance between the circles and the triangle indicates current possesion of the resource.

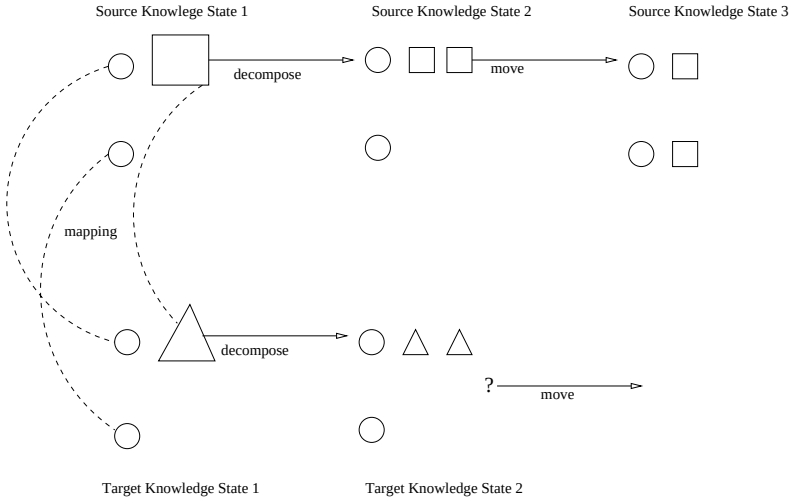


Fig. 10. This figure illustrates a state of problem solving during transfer of a problem-solving procedure from a source case to the hypothetical problem. In this state, the `decompose transformation` has just been transferred from the source case and applied to the first `s-image` in the target problem. Th application of this transformtion creates a new element in the second `s-image of the target case.`

`s-image` representing the target problem and the first `s-image` in the source case.

Now consider what happens when the problem-solving procedure is transferred from the source case to the target problem. First, the `decompose transformation` in the source case is transferred to the target and applied to its `s-image`. This results in the generation of an intermediate knowledge state containing two smaller triangles. The difficulty arises when the next `transformation`, `move`, is transferred from the source case and applied to the newly generated `s-image` in the target problem: what element in the `s-image` should `move` apply to? Since the small triangles in the target are the creation of the previous `transformation`, the initial mapping between the source and the target does not have any mapping between the small triangles in the target and the small squares in the source!

Therefore, it follows that if and when a `transformation` in the target case creates a new object, there is a need to dynamically generate a new mapping between the newly created intermediate knowledge state in the target and the corresponding intermediate knowledge state in the source case. Note that although we have described this problem in the context of visual analogy in problem solving, it appears to be independent of the type of knowledge; instead, it seems like a characteristic of all analogical problem solving in which new objects are created.

2.4.3 Transfer Algorithm

Table 11 contains a more formal specification of the main algorithm for the transfer task. The unspecified functions used in this algorithm (e.g., adapt-arguments, carry-over-unchanged-relationships) are described in the following subsections.

2.4.4 Adapt-arguments

When an argument needs to be adapted to the target problem, Proteus determines whether it is a `literal`, a `function`, or a `component` of an `s-image.` Literals are returned verbatim. If the argument is a function (e.g. the number of people in a group) then Proteus applies the same function to the analogous group in the target and returns that value. If the argument is a component, then Proteus returns the analogous object in the target. The algorithm apears in table 12.

In the fortress/tumor problem, the `adapt-arguments` algorithm takes in the symbols `FOUR` and `FORTRESS-PROBLEM-TUMOR-PROBLEM-MAPPING1`. Since `FOUR` is in Galatea's list of literals, it executes the "literal" case and returns the symbol as is: `FOUR.`

Since this case does not occur in the fortress/tumor problem, we will use the cake/pizza example to describe it. The reasoner needs to feed six people with one Sicilian slice sheet pizza. An analog in memory of cutting a sheet cake for four people is used to generate a solution. Transfer is still difficult because somehow the `Four` in the cake analog must be adapted to the number `Six` in the pizza analog. Knowing how many pieces into which to cut the cake or pizza depends on the number of people in each problem. Some notion of count is needed. The use of `functions` as arguments to `transformations` addresses this problem. The cake analog is represented with a `function` that counts the number of people as its argument for the `decompose transformation`. This function has an argument of its own, namely the set of cake eaters, which during adaptation adapts into the set of pizza eaters. When the `transformation` is applied to the pizza, it counts the members of the set of people in the pizza

function main-algorithm

Input:	1. Source 2. Target problem 3. Vertical mapping between source and target
Output:	1. A set of new target knowledge states 2. Vertical mappings between corresponding source and target states 3. Horizontal mappings between successive target states 4. Transformations connecting successive target states

Procedure

1: **while** more-source-states(goal-conditions, memory) **do**
2: current-s-image ← get-next-target-s-image(target problem, current-s-image)
3: current-source-s-image ← get-next-source-s-image(source, current-source-s-image)
4: current-transformation ← get-transformation(current-s-image)
5: current-arguments ← get-arguments(current-source-s-image)
6: source-objects-of-transformation ← get-target-object-of-trans(current-source-s-image)
7: current-vertical-mapping ← get-mapping(current-target-s-image, current-source-s-image)
8: target-object-of-transformation ← get-source-object-of-transformation(current-vertical-mapping, source-objects-of-transformation)
9: target-arguments ← adapt-arguments(get-arguments(current-source-s-image, current-source-s-image))
10: memory ← memory + apply-transformation(current-transformation, target-object-of-transformation, target-arguments)
11: memory ← memory + create-horizontal-mapping(current-target-s-image, get-next-target-s-image)
12: current-target-s-image ← get-next-target-s-image
13: current-source-s-image ← get-next-source-s-image
14: memory ← memory + carry-over-unchanged-relationships(applied-transformation)
15: memory ← memory + create-vertical-mapping(current-target-s-image, current-source-s-image)

Table 11
Main algorithm

function adapt-arguments

Input:	1. argument 2. mapping
Output:	1. an adapted argument.

Procedure:

```
1: if literal? argument then
2:     return argument
3: else if function? argument then
4:     return calculate-function(argument)
5: else if component? argument then
6:     return (get-analogous-component(argument, mapping))
```

Table 12
Algorithm for adapt-arguments

problem (which results in six). `Decompose` produces six pieces of pizza in the next `s-image`.

2.4.5 Carry-over-unchanged-relationships

In table 13 is a description of the `carry-over-unchanged-relationships` function. The `get-analogous-chunks` sub-function constructs returns chunks that are identical to the input chunks, except that the symbols that have `maps` in the input `mapping` are replaced with those symbols they are associated with in those `maps`. The vertical `map` relationships are carried over as well, constituting the vertical maps for unchanged components.

2.4.6 Creation-of-horizontal-maps-between-changed-components

The `creation-of-horizontal-maps-between-changed-components` (see table 14) is embedded in the code for each of the `transformations`. The `transformation` results are obtained from running the `transformation`. The `target-objects-of-transformation` are known because they are the input to the `transformation`. The two lists are put in alphabetical order and `maps` are created between each *nth* list object. These are `maps` within a procedure, showing what `elements` in earlier `s-images` turn into what elements in later `s-images`.

Similarly, `creation-of-horizontal-maps-between-unchanged-components` (see table 15) makes `maps` between old objects (the objects in the `old-s-image` and new objects (from the `current-s-image`, minus the objects created by the `transformation`), alphabetizes them, and creates maps between the *nth* item in each list.

function carry-over-unchanged-relationships

Input:	1. The Memory: memory 2. The horizontal mapping: h-mapping 3. Transformation 4. Previous-s-image
Output:	1. Analogous chunks.

Procedure:

1: new-chunks ← get-chunks(run-transformation(transformation))
2: old-analogous-chunks ← get-analogous-chunks(new-chunks, h-mapping)
3: old-chunks ← get-all-chunks(previous-s-image)
4: chunks-to-transfer ← old-chunks − old-analogous-chunks
5: memory ← memory + create-analogous-chunks(chunks-to-transfer, h-mapping)

Table 13
Algorithm for carrying over unchanged relationships

function creation-of-horizontal-maps-between-changed-components

Input:	1. Transformation results 2. Target-objects-of-transformation
Output:	1. New horizontal maps between the current and next s-image.

Procedure:

1: post-transform-components ← get-chunks(run-transformation(transformation))
2: memory ← memory + create-maps(post-transform-components, target-objects-of-transformation)

Table 14
Algorithm for creation of horizontal maps between changed components

2.4.7 Creation-of-vertical-maps-between-changed-components

The algorithm for creating vertical maps between changed components (see table 16) takes as input the transformation results in the source *and* target, alphabetizes them, and creates `maps` between the *nth* item in each list.

2.5 Evaluation of the Transferred Solution

We have tried without sucesss to come up with a method for fully automated run-time evaluation of the transferred solution to the target problem using

function creation-of-horizontal-maps-between-unchanged-component

Input:	1. Transformation results 2. Old-s-image 3. Current-s-image 4. Post-transform-components 5. Old-components 6. Current-components
Output:	1. new horizontal maps between the current and next s-image.

Procedure:

1: old-components ← get-all-components(old-s-image) − target-objects-of-transformation
2: current-components ← get-all-components(current-s-image) − post-transform-components
3: memory ← memory + create-maps(old-components, current-components)

Table 15
Algorithm for creation of horizontal maps between unchanged components.

function creation-of-vertical-maps-between-changed-components

Input:	1. Target transformation results 2. Source transformation results 3. New-target-components 4. New-source-components
Output:	1. new vertical maps between the current source and `s-images`.

Procedure:

1: new-target-components ← target transformation results
2: new-source-components ← source transformation results
3: memory ← memory + create-maps(new-target-components, new-source-components)

Table 16
Algorithm for creation of vertical maps between changed components.

only visual knowledge. Thus, we suggest that the evaluation subtask of visual analogy in problem solving necessarily requires explicitly represented causal and functional knowledge. Of course, in some cases, it might be possible to derive the causal and functional knowledge from the visual representation but that is beyond the scope of the Proteus project.

Thus, Proteus currently uses precompiled knowledge for evaluating the can-

didate solution to the target problem. As Proteus' designers, we handcrafted the propositions that a correct solution should have generated and compiled them into the evaluation task. When Proteus proposes a candidate solution, it compares the propositions generated by the candudate with the propositions generated by the correct solution. If the candidate solution fails, then Proteus returns to the output of retrieval task, and selects a different source case (if one is available) and attempts to transfer its procedure to the target problem.

3 Evaluation of Proteus

Proteus is an integration of two systems: Geminus [43] and Galatea [5,7,6]. The Geminus subsystem of Proteus is responsible for the retrieval, mapping and storage tasks, the Galatea subsystem is responsible for the transfer and evaluation tasks, including dynamic generation of mappings between the intermediate knowlede states in the source and target cases. Some of the evaluation of Proteus has been in the context of the Geminus and Galatea subsystems.

Uniformity: Proteus uses an uniform knowledge representation language, Covlan, for all tasks and subtasks of visual analogy in problem solving.

Generality: We have validated different parts of Proteus for a large range of problems. In particular, we have validated Geminus' retrieval method for a variety of 2-D, vector graphic, line drawings. Similarly, we have validated Galatea's transfer method for problems ranging in complexity from cutting a simple circular shape (e.g., a pizza) in analogy to cutting a similar shape (e.g., a cake) into smaller parts, to analogy-based design of a weed-trimmer, to a historical case study of James Clerk Maxwell's reasoning about vortices in electromagnetic fields [7]. We have validated Proteus itself for both the pizza-cake problem and the fortress-tumor peoblem.

The choice of the fortress/tumor problem as a running example in this paper has been deliberate: The fortress/tumor problem is often considered to be a cannonical example in the literature on analogy (e.g., [21]) and because earlier computational models of the fortress/tumor classical example have relied on the use of (non-visual) causal and functional knowledge [30]. Thus, successful execution of Proteus on the fortress/tumor problem partly confirms our initial hypothesis: visual knowledge alone is sufficient for analogical transfer of problem-solving procedures in some task domains. It also leads to a refinement of the initial hypothesis: while visual knowledge is sufficient for the retrieval, mapping, transfer and storage subtasks of analogy, the evaluation subtask appears to require (non-visual) causal and functional knowledge.

Efficiency: We have conducted efficiency experiments with Geminus. In our experiments, the long-term memory contained 42 source cases; the number of visual elements in the indexical `s-images` of the source cases ranged from 3 to over 50, with an average about 12' and the number of propositions in semantic network representing the `s-images` ranged from a couple of dozen to over eight thousand. The experiments were conducted with 21 target problems; each of the `s-images` in the targer problems contains 2 to 5 visual elements and up to several dozen propositions. Running on a desktop workstatation, Geminus retrieved the relevant `s-images` in about 9.32 seconds on average across all 21 target `s-images`), doing an average of about 1.49 million memory accesses (to the index of propositions across all the source `s-images`) per retrieval. Even in the worst-case (characterized by the size of the target `s-image`, Geminus took under a minute to retrieve the relevant source `s-images`.

Recall that following MAC/FAC [17] the Geminus decomposes the retrieval task into two subtasks: reminding and selection. Recall also that following ACME [31], Geminus uses a constraint-satisfaction method for the selection task. Ablation experiments with Gemimus [43], in which we removed the reminding subtask of the retrieval task and performed retrieval based solely on contraint satisfaction, revealed no significant degradation in its performance. This leads us to the following conjecture about case retrieval in general: when constraint-satisfaction is used for selection, there may be little need for reminding using feature vectors. This conjecture needs further examination.

Cognitive Modeling: We have used Galatea to model the input-output behavior of human subjects engaged in analogy-based design [6]. In these experiments [3], human subjects were given a source case of a design of the entrance to a clean-air laboratory, where the problem was articulated in text and the solution was expressed both in text and in a drawing. The source case also contained a problem-solving procedure articulated in text form, which converted an entrance with a single door to a vestibule with double doors. In the target problem, which was expressed in text form, the subjects were asked to draw the design a portable weed-trimmer in analogy to the entrance to the clean-air laboratory. We then used Galatea to solve exactly the same target problem with exactly the same source case, except that in Galatea both the source case and the target problem were expressed only visually. We found that Galatea successfully solved the above problem.

For four of the human subjects in the above study, we conducted an additional experiment with Galatea. We ran Galatea under different initial knowledge conditions, but without any change to its computational process, its knowledge representation language, or its algorithms. We found that by simply changing the intial knowledge conditions, we could make Galatea replicate

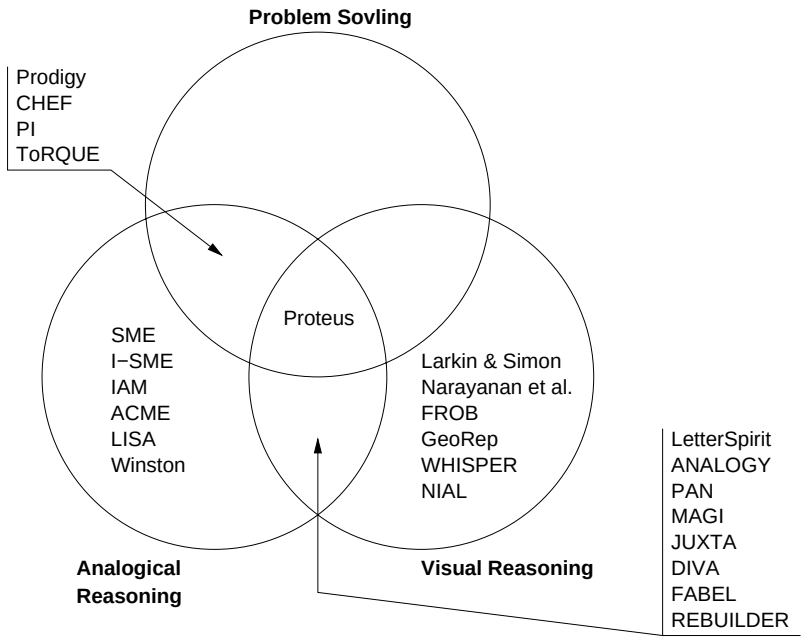


Fig. 11. Proteus lies at the intersection of problem solving, analogical reasoning and visual reasoning. This Figure depicts its relationshp with other theories and systems.

about half of the features in the drawings made by the four human subjects. The other half, which related to numerical dimensions and causal processes, however are beyond Galatea. this paper no part of Proteus is proposed as a cognitive model. The Galatea section simply replicates some elements in the input-output behavior of human subjects engaged in analogy-based design. The above experiments however indicate the range of problems it can address. Determining whether the level of abstraction of Proteus/Galatea's visual knowledge is consistent with that of human designers requires additional research.

4 Related Work

Figure 11 indicates that Proteus lies at the intersection of problem solving, analogical reasoning and visual reasoning: at the task level, it addresses problem solving; at the method level, it uses analogies to solve problems; and in terms of knowledge, it uses only visual knowledge. The Figure also shows where related theories and systems lie on the same Venn diagram.

As we mentioned earlier, Proteus' decomposition of the retrieval task into feature-based reminding and structure-based selection follows MAC/FAC [17]. However, while MAC/FAC uses structure-mapping for the selection task, Proteus uses constraint satisfaction with backtracking. As we noted earlier, the experiments with the Geminus subsystem of Proteus indicate that the two-stage decomposition of the retrieval task provides little computational benefit over just one-stage retrieval based on constraint satisfaction [43] .

The Structure-Mapping Engine [10] is based on cognitive model of Structure-Mapping [20]. It constrains the mapping problem with several rules including the systematicity principle, which holds that higher-order (more nested) relational similarities are preferred over lower-order (less nested) similarities. SME finds many possible mappings between a source and a target, then evaluates them according to the map rules. SME however neither provides a content account of visual analogies, nor does it address problem solving, i.e., it transfers relations, not procedures.

I-SME [16] and the Incremental Analogy Machine (IAM) [33] are incremental mappers. An incremental mapper generates a mapping as objects in a given case are introduced to the mapper one at a time. Proteus does incremental mapping of a different kind and at a different stage of processing: if and when an operation creates a new object during transfer of a problem-solving procedure it dynamically generates new `mappings` for the newly created objects.

As we mentioned earlier, Proteus' view of retrieval as a constraint satisfaction problem follows that of ACME [31]. ACME (for Analogical Constraint Mapping Engine) uses structural, semantic, and pragmatic constraints for mapping. Structure, in this sense, does not necessarily mean physical appearance, but the nature of the representation: elements are structurally similar if they share the same relational structure with other elements. Semantic similarity refers to elements are either identical symbols or share predicates (e.g. a common superordinate). Pragmatic constraints pertain to the relative importance of some propositions in the representation given the goals of the agent. The mapping is generated as a result of a constraint-satisfaction spreading activation network. Transfer in ACME involves transferring relations and postulating new elements from the source analog. ACME neither provides a content account of visual analogy, nor does it address problem solving, i.e., it transfers facts, not procedures.

LISA [32] is cognitive model of analogical mapping. Propositions are made up of units spread activation to each other. Arguments of propositions fire in synchrony with the case roles to which they are bound, and out of synchrony

with other case roles and arguments. Through spreading activation, the best map is found.

Winston [42] describes an analogical mapper based on content account of its domain (understanding political stories). The content account is based on causality and functionality. The mapper generate all possible mappings, and then ranks them. While Proteus' content and process accounts are very different from that of Winston, like Winston's work, Proteus too emphasizes the importance of interaction between knowledge content and reasoning processes.

4.2 Analogical Problem Solving

CHEF [28] is a case-based reasoner that adapts cooking recipies from a source to a target. CHEF uses non-visual causal and functional knowledge. Further, it neither represents intermediate knowledge states, nor transfers procedures that create new objects.

The Derivational Analogy theory, [41,40,38] implemented in the Prodigy system, analogically transfers problem-solving procedures from source cases to target problems. It uses traces of problem solving, called *derivations*, for enabling the transfer. Derivations are scripts of the steps of problem solving, which contain justifications for why the specific steps in problem solving were chosen over others. Prodigy also allows for adaptation of the transferred procedure. Like CHEF, Prodigy too uses non-visual causal and functional knowledge. Further, in Prodigy, the intermediate knowledge states are not explicitly represented or saved in the case memory. Instead, a stored case contains only the record of the changes made to the states, which allows the knowledge states to be inferred. In contrast, Proteus explicitly represents intermediate knowledge and stores them in the case memory. Furthemore, whenever an operation in the transferred procedure creates new objects, Proteus dynamically generates new mapping between the intermediate knowledge states in the target and the source case. Prodigy is able to avoid generation of these intermediate mappings because the problems with which it deals do not have procedures that create new objects.

The Process of Induction (PI) model [30] is the only implemented computational model, other than Proteus, that solves the fortress/tumor problem. However, unilke Proteus, it uses non-visual causal and functional knowledge. PI does not represent intermediate knowledge states or generate new mappings between them. This is because it does not do analogical transfer in the usual sense of the term. Instead, it first uses a production system to solve the fortress problem. Once PI solves the tumor problem, it induces an abstract schema that works as a single rule that aapplies to both problems.

The ToRQUE2 system [25–27] uses a taxonomy of generic structural transformations that can be applied to physical systems . These transformations were found to be useful in modeling protocols of human subjects solving a problem dealing with spring systems. ToRQUE2's structural representations are different from Proteus' visual representations: the structural representations describe a system's physical composition but typically include only the information directly relevant for predicting the causal behaviors of the system. Structural knowledge, like a schematic, shows the components of the system and the connections among them but leaves out other visual information, such as what a component wire looks like, which side of a pump is up, etc.

4.3 Visual Knowledge and Reasoning

Larkin and Simon [34] describe a system that reasons about physical systems such as pulley systems. In their representation, objects are not represented explicitly, but implicitly through locations: when one location is attended to, all information there is attended to. To answer a question about a pulley system, the reasoner uses some non-visual causal knowledge of physics along with the visual knowledge.

The NIAL system [23] distinguishes between *depictive* and *descriptive* representations (corresponding to bitmape and propositional representations), as well as a distinction between *visual* and *spatial* (corresponding to *where* something is and *what* something is). The descriptive representation is stored in memory, and the depictive is generated in a working memory when needed.

WHISPER [18] is a problem-solving system that can request observations from and make changes to depictive diagrams of a blocks world. It knows about stability and falling objects. It can visualize something rotating in the diagram and determine when it will hit another object. The system's goal is to move all blocks until they are stable. It moves them, then simulates how they will act (in a bitmap "retina") for evaluation. While WHISPER uses visual knowledge for many of its tasks, it uses non-visual causal knowledge for evaluation by simulation.

The FROB system [15] uses Qualitative Spatial Reasoning (QSR) [14] to reason about physical systems, e.g., figuring out the paths of balls on a landscape and whether will collide. In QSR, an agent needs both a metric diagram representation and a place vocabulary representation. A metric diagram shows the quantitative aspects of the system, like sizes of objects expressed in numbers. The place vocabulary is a qualitative representation of location and shapes of objects. A place is a region of space where some important property (e.g. being in contact with something) is constant.

GeoRep [13] takes in 2-D line drawings and outputs the visual relations in it. First it uses a LLRD (low-level relational describer) module to aggregate visual primitives. Its visual primitives are: line segments, circular arcs, circles, ellipses, splines, and text strings. It finds relations of the following kinds: grouping, proximity detection, reference frame relations, parallel lines, connection relations, polygon and polyline detection, interval relations, and boundary descriptions. Then the HLRD (high-level relational describer) finds higher-level, more domain-specific primitives and relations. GeoRep's content account is at the low level – the higher level account is left up to the modeler. Although though the reasoning goal in Proteus is very different from that of GeoRep, its knowledge representation language, Covlan, has considerable overlap with GeoRep's vocabulary. Covlan, however, also provides a vocabulary for representing visual operations, such as `move-to-location`, and for representing a sequence of visual operations in a procedure.

Narayanan, Suwa and Motoda's model [36] describe a cognitive model for predicting the behavior of physical systems from their diagrams. The visual knowledge is represented with diagram frames (representing lines and spaces and connections between them) and occupancy array representations (representing, for each pixel, what kind of object is located there). The vocabulary of Proteus' knowledge representation language is at the same level of abstraction as Narayanan, Suwa and Motoda's model.

4.4 Visual Analogy

We have already described the ANALOGY and Letter Spirit systems in the introduction. A couple of addtional remarks about Letter Spirit are in order. Letter Spirit transfers single transformations/attributes (e.g. crossbar-suppressed) and therefore cannot make analogical transfer of procedures (e.g. moving something, then resizing it) which Proteus can do. In contrast, one can see how Proteus might be applied to the Letter Spirit's font domain: The stylistic guidelines in LetterSpirit, such as "crossbar suppressed" are like the visual transformations in Proteus: "crossbar supressed would be a transformation of removing an element from an `s-image`, where element removed would the crossbar and the `s-image` would be the prototype letter *f*. Once the transformation is expressed in Proteus' vocabulary, it could be applied to all the other letters one by one. In this way, Proteus is more general than LetterSpirit.

Like ANALOGY, the PAN system [37] uses graph-like representations of abstract diagrams and outputs transformations that will turn one diagram into another. Neither ANALOGY nor PAN however can transfer problem-solving procedures.

MAGI [11] takes visual representations and uses the Structure-Mapping Engine to find examples of symmetry and repetition in a single diagram. JUXTA [12] uses MAGI in its processing of a diagram of two parts. It outputs a mapping between the images, and notes distracting and important differences. Both MAGI and JUXTA use GeoRep's visual representation language.

DIVA [4] is another analogical mapper that uses visual representations. Specifically, it uses the Java Visual Object System. its examples is the fortress/tumor problem, however, it does not transfer problem solving procedures.

In computer-aided design, FABEL [19] was an early project to explore the automated reuse of diagrammatic cases. In particular, TOPO [2], a subsystem of FABEL, used the maximum common subgraph (MCS) of the target drawing with the stored drawings for retrieve similar drawings.

REBUILDER [24] is a case-based reasoner that does analogical retrieval, mapping, and transfer of software design class diagrams. The diagrams are represented structurally, not visually, however. This means that while REBUILDER may represent that a teacher has a relationship with a school for example, it does not represent as any left-of/right-of, above/below connection between them.

5 Conclusion

We have described a computational theory of visual analogy in problem solving that addresses all major subtasks of visual analogy. Proteus is a computer program that implements and substantiates the theory. The contributions of this paper are that (a) for the first time it describes Proteus as a whole, and (b) it provides a detailed account of the knowledge representations and algorithms used by Proteus. Proteus provides two main things: (a) a content account and (b) a process account for visual analogy in problem solving. Our work on Proteus leads us to the following three conclusions:

I: Visual knowledge alone is sufficient for some subtasks of analogical problem-solving.

We started this work with the hypothesis that visual knowledge alone is sufficient for analogical problem solving. We deliberately chose the fortress/tumor problem to test this hypothesis not only because it is a classic example of analogical problem solving but also because previous computational models for this problem, such as PI [30], relied solely on non-visual causal and functional knowledge. Based on Proteus, we now refine this hypothesis: visual knowledge alone is sufficient for the retrieval, mapping and transfer subtasks of analogical

transfer of problem-solving procedures.

Proteus also shows that while visual knowledge is useful for the mapping task, it alone is not suffiicent for generating a single mapping between a target problem and a retrieved source case matching the target. The difficulty is that while visual knowledge captures structural constraints (e.g., shapes, and spatial relations), it does not capture causal and functional knowledge of either the source or the target, which leaves the mapping task underconstrained and results in the generation of multiple mappings between the target and the source. Note that this finding is consistent with earlier work on visual analogy: while ANALOGY does do mapping based on visual knowledge alone, it does so in a context entirely different from and much simpler than transfer of problem-solving procedures, and, in Letter Spirit, the mapping is compiled into the system's knowledge.

In addition, Proteus shows that visual knowledge alone cannot enable evaluation of the transferred solution for the target problem. This finding is consistent with other work on visual reasoning work, such as WHISPER [18], which uses non-visual causal knowledge for evaluation.

> *II: Visual analogy is enabled by knowledge of shapes, spatial relations, and shape and spatial transformations.*

Proteus uses Covlan, an uniform knowledge representation language, for all subtasks of visual knowledge. In particular, Proteus shows that the retrieval task of visual analogy is enabled by symbolically represented knowledge of shapes of objects (e.g., spline, ellipse) and spatial relations between the shapes (e.g., above, left-of) in drawings representing the target and source cases. It also shows that the transfer task requires additional knowledge, namely, symbolic knowledge of transformations of shapes and spatial relations from one knowledge state to the next (e.g., move, move-set). The latter is required because of the need to represent the problem solving steps and their effects. Of course, Covlan is incomplete; a larger class of domains may require additions of its vocabulary.

> *III: Successful analogical transfer of strongly-ordered procedures in which new objects are created requires generation of new mappings between the intermediate knowledge states in the source and target cases.*

An unexpected finding of our work on Proteus is that that the successful transfer of strongly-ordered procedures *in which new objects are created* requires the problem-solver to dynamically generate intermediate knowledge states and `new mappings` between the intermediate knowledge states of the source and target analogs. This is because the newly created objects are acted on by later operations. In the tumor problem, for example, the strong ray is first turned into a several weaker rays. When the problem solver transfers the move solid-

ers operation from the fortress case and seeks to apply the move operation to the tumor problem, how does it know that the objects corresponding to the soliders are the weaker rays? It *must* have some mapping to make this inference, and since the weaker rays do not exist in the start state of the tumor problem, this mapping cannot be part of the initial mapping between the target and the source. Therefore, the new knowledge state with the weaker rays must be generated, and then a new `mapping` must be made between the new knowledge state and the corresponding knowledge state in the source case.

Acknowledgements

Nancy Nersessian has been a collaborator on the work on Galatea subsystem of the Proteus system.

References

[1] M. Anderson, B. Meyer, and P. Olivier, editors. *Diagrammatic Representation and Reasoning*. Springer, 2002.

[2] K. Börner, P. Eberhard, E.-C. Tammer, and C.-H. Coulon. Structural similarity and adaptation. In I. Smith and B. Faltings, editors, *Advances in Cased-Based Reasoning: Proc. 3rd European Workshop on Cased-Based Reasoning*, volume 1168 of *Lecture Notes in Artificial Intelligence*, pages 58–75, Lausanne, Switzerland, 1996. Springer-Verlag.

[3] D. L. Craig, R. Catrambone, and J. Nersessian, Nancy. Perceptual simulation in analogical problem solving. In *Model-Based Reasoning: Science, Technology, & Values*, pages 167–191. Kluwer Academic / Plenum Publishers, 2002.

[4] D. Croft and P. Thagard. Dynamic imagery: A computational model of motion and visual analogy. In L. Magnani and N. J. Nersessian, editors, *Model-Based Reasoning: Science, Technology, & Values*, pages 259–274. Kluwer Academic: Plenum Publishers, 2002.

[5] J. Davies and A. K. Goel. Visual analogy in problem solving. In *Proceedings of the International Joint Conference for Artificial Intelligence 2001*, pages 377–382. Morgan Kaufmann Publishers, 2001.

[6] J. Davies, A. K. Goel, and N. J. Nersessian. Transfer in visual case-based problem-solving. In *Proceedings of the 6th International Conference on Case-Based Reasoning*. Springer-Verlag, 2005.

[7] J. Davies, N. J. Nersessian, and A. K. Goel. Visual models in analogical problem solving. *Foundations of Science*, 2002. special issue on Model-Based Reasoning: Visual, Analogical, Simulative.

[8] K. Duncker. A qualitative (experimental and theoretical) study of productive thinking (solving of comprehensible problems). *Journal of Genetic Psychology*, pages 642–708, 1926.

[9] T. G. Evans. A heuristic program to solve geometric analogy problems. In M. Minsky, editor, *Semantic Information Processing*. MIT Press, 1968.

[10] B. Falkenhainer, K. D. Forbus, and D. Gentner. The structure-mapping engine: Algorithm and examples. *Artificial Intelligence*, 41:1–63, 1990.

[11] R. W. Ferguson. Magi: Analogy-based encoding using regularity and symmetry. In *Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society*, pages 283–288, 1994.

[12] R. W. Ferguson and K. D. Forbus. Telling juxtapositions: Using repetition and alignable difference in diagram understanding. In K. Holyoak, D. Gentner, and B. Kokinov, editors, *Advances in Analogy Research*, pages 109–117. New Bulgarian University, 1998.

[13] R. W. Ferguson and K. D. Forbus. Georep: A flexible tool for spatial representation of line drawings. *Proceedings of the 18th National Conference on Artificial Intelligence*, 2000.

[14] K. D. Forbus. Qualitative kinematics: A framework. In D. S. Weld and J. de Kleer, editors, *Readings in Qualitative Reasoning About Physical Systems*, pages 562–567. Morgan Kaufmann Publishers, San Mateo, California, 1990.

[15] K. D. Forbus. Qualitative spatial reasoning framework and frontiers. In J. Glasgow, N. H. Narayanan, and B. Chandrasekaran, editors, *Diagrammatic Reasoning: Cognitive and Computational Perspectives*, pages 183–202. AAAI Press/MIT Press, 1995.

[16] K. D. Forbus, R. W. Ferguson, and D. Gentner. Incremental structure-mapping. In *Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society*, pages 313–318, 1994.

[17] K. D. Forbus, D. Gentner, and K. Law. MAC/FAC: A model of similarity-based retrieval. *Cognitive Science*, 19(2):141–205, 1995.

[18] B. V. Funt. Problem-solving with diagrammatic representations. *Artificial Intelligence*, pages 201–230, 1980.

[19] F. Gebhardt, A. Voss, W. Grather, and B. Schmidt-Belz. *Reasoning with Complex Cases*. Kluwer, 1997.

[20] D. Gentner. Structure-mapping: A theoretical framework for analogy. *Cognitive Science*, 7(2):155–170, 1983.

[21] M. L. Gick and K. J. Holyoak. Analogical problem solving. *Cognitive Psychology*, pages 306–355, 1980.

[22] J. Glasgow, H. Narayanan, and B. Chandrasekaran, editors. *Diagrammatic Reasoning: Cognitive and Computational Perspectives*. AAAI Press/MIT Press, 1995.

[23] J. Glasgow and D. Papadias. Computational imagery. In P. Thagard, editor, *Mind Readings*. MIT Press, 1998.

[24] P. Gomes, N. Seco, F. C. Pereira, P. Paiva, P. Carreiro, J. L. Ferreira, and C. Bento. The importance of retrieval in creative design analogies. In *Creative Systems: Approaches to Creativity in AI and Cognitive Science. Workshop program in the Eighteenth International Joint Conference on Artificial Intelligence*, pages 37–45, 2003.

[25] T. W. Griffith. *A Computational Theory of Generative Modeling in Scientific Reasoning*. PhD thesis, College of Computing, Georgia Institute of Technology, 2000.

[26] T. W. Griffith, N. J. Nersessian, and A. K. Goel. The role of generic models in conceptual change. In *Proceedings of the Eighteenth Annual Conference of the Cognitive Science Society*, pages 312–317, 1996.

[27] T. W. Griffith, N. J. Nersessian, and A. K. Goel. Function-follows-form transformations in scientific problem solving. In *Proceedings of the Twenty-second Annual Conference of the Cognitive Science Society*, 2000.

[28] K. J. Hammond. Case-based planning: A framework for planning from experience. *Cognitive Science*, pages 385–443, 1990.

[29] K. J. Holyoak and P. Thagard. Analogical mapping by constraint satisfaction. *Cognitive Science*, pages 295–355, 1989.

[30] K. J. Holyoak and P. Thagard. A computational model of analogical problem solving. In S. Vosniadou and A. Ortony, editors, *Similarity and analogical reasoning*, pages 242–266. Cambridge University Press, 1989.

[31] K. J. Holyoak and P. Thagard. The analogical mind. *American Psychologist*, pages 35–44, 1997.

[32] J. Hummel and K. J. Holyoak. Lisa: A computational model of analogical inference and schema induction. In *Proceedings of the Eighteenth Annual Conference of the Cognitive Science Society*, pages 352–357, 1996.

[33] M. T. Keane and M. Brayshaw. The incremental analogy machine. In *Proceedings of the Third European Working Session on Learning*, pages 53–62, 1988.

[34] J. Larkin and H. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, pages 65–99, 1987.

[35] G. McGraw and D. R. Hofstadter. Perception and creation of alphabetic style. Technical report, AAAI, 1993.

[36] H. N. Narayanan, M. Suwa, and H. Motoda. How things appear to work: Predicting behaviors from device diagrams. In *Proceedings of the 12th National Conference on Artificial Intelligence*, pages 1161–1167, 1994.

[37] S. O'Hara and B. Indurkhya. Incorporating (re)-interpretation in case-based reasoning. In *Proceedings of the First European Workshop on Case-Based Reasoning (EWCBR-93)*, pages 246–260, 1994.

[38] U. Schmid and J. Carbonell. Empirical evidence for derivational analogy. In *Proceedings of the 21st Annual Conference of the Cognitive Science Society*, page p. 814, 1999.

[39] P. Thagard, K. J. Holyoak, G. Nelson, and D. Gochfeld. Analog retrieval by constrain satisfaction. *Artificial Intelligence*, 46:259–310, 1990.

[40] M. M. Veloso. Prodigy/analogy: Analogical reasoning in general problem solving. In *EWCBR*, pages 33–52, 1993.

[41] M. M. Veloso and J. G. Carbonell. Derivational analogy in prodigy: Automating case acquisition, storage, and utilization. *Machine Learning*, pages 249–278, 1993.

[42] P. H. Winston. Learning and reasoning by analogy. *Communications of the ACM*, 1980.

[43] P. W. Yaner and A. K. Goel. Visual analogy: Viewing analogical retrieval and mapping as constraint satisfaction problems. accepted for publication, 2005.