Analogical Mapping Through Visual Abstraction

Jim Davies (jim@jimdavies.org)
Science of Imagination Laboratory
Institute of Cognitive Science; Carleton University; 1125 Colonel By Drive
Ottawa, Ontario, K1S 5B6 Canada

Patrick W. Yaner (patrick.yaner@gmail.com)
Artificial Intelligence Laboratory
College of Computing; Georgia Institute of Technology; 801 Atlantic Drive
Atlanta, GA 30332-0280 USA

Abstract

Analogical mapping theories tend to focus on matching identical symbols (either for objects or the relations between them). In the domain of visual representations we implemented a mapping system that uses separate domain knowledge (a shape-type superclass hierarchy) to re-represent analogs such that identicality can be found at different levels of abstraction. Such a scheme is useful where shape, and not the spatial layout of the analog images, is important to aligning visual objects.

Introduction

Mapping, a core part of analogy, is finding the alignments between the elements of two analogs. For example, in an analogy between a face and the front of a car, a mapping might include an alignment between the eyes and the windshield. Such an alignment might be based on the fact that the "perceptual" input to the car happens at the windshield for a car as do the eyes for a face.

Mapping is combinatorially complex, and this complexity is reduced by finding similarity between the elements to be mapped. In the example above, the alignment is justified by the *functional* similarity between the eyes and the windshield. In the discussion section we will describe different similarity measures for implemented mapping systems.

Our work focuses on mapping for purely *visual* analogs. That is, we are exploring different similarity measures that are appropriate for mapping the visual components of images. To return to the car/face example, rather than *functionally* aligning the eyes and the windshield, an agent that focused on visual similarity might align the eyes to the headlights because they both consist of two elements, are both round, and are horizontally oriented.

One way to find similarity between visual elements is by identification of identical symbols relating the elements. Most simply, if two elements are described in the representation as *square*, then the mapping agent can favor their alignment. Another way is to identify identical symbols that *relate* elements of the same image. For example, if in one image *element-x* is related to *element-y* with a symbol *is-above*, then a mapper interested in the structure of images might want to align x and y to f and g, if indeed f *is-above* g in the other analog—regardless of what shape x, y, f and g are. Structure Mapping Theory (Gentner, 1983) uses identicality of the symbols describing structure to find mappings between analogs.

Grouping

The Gestalt psychologists found that people perceptually grouped visual elements according to, among other aspects, shared orientation, color, and proximity. This provides psychological evidence for an explicit representation of visual element grouping. Broadly speaking there are two kinds of groups in our work: *aggregations* and *sets*. Aggregations are multiple visual elements that form one coherent shape (e.g. a square is an aggregate of four lines). Sets are groups of elements that are unconnected but similar in some way (e.g. nuts in a bowl).

Sets and aggregates appear at a certain level of abstraction, at which they can be aligned to each other as visual elements. An agent with a flexible representation can, however, zoom into these groups. There are two reasons an agent might want to do this: First, If the agent must decide which group to align to which other group, the conflict resolution might require an examination of the contents of that group. Second, it might be important to align group members. For example, imagine aligning two armies—at this level of abstraction armies can be moved and split apart, and it makes sense to have the armies' generals simply be members of the army sets. But if the agent needs to reason about the generals in particular, it could be important to know that the general in one set aligns to the general in the other. In analogical problem solving, for example, certain operations need to be applied to elements of analogs.

Different levels of representation are needed for different transformations applied to the analog (Davies, Goel, & Nersessian, 2003). Likewise with aggregates, mapping one box to another is the right level of abstraction for motion of the entire set, but opening one side of the box by moving one of its constituent lines requires a mapping at the component level.

For these reasons it is helpful for an agent to be

flexible in its representations such that it can reason at multiple levels of grouping abstraction.

Shape-type Superclass Hierarchy

The above similarity notions use the nature of the analogs as given in the representation. However similarity can also be found through the application of domain knowledge to the analogs. In the visual domain, this can take the form of a shape-type hierarchy (See Figure 2.) For example, a right triangle and an isosceles triangle are similar because they are both triangles. Even in cases where the term *triangle*, and its relation to *right-triangle* and *isosceles-triangle* are not explicit in the representation of the analogs, an agent can use the domain knowledge of a shape-type hierarchy to find element similarity. We will show that abstraction using this hierarchy is particularly useful (compared to structure-mapping) for analogs in which the spatial arrangement of the visual elements is less important than the shapes of the objects represented. The examples we explicate below are of this type. Using abstraction has been used in Minimal Ascension (Falkenhainer, 1988) and in cross-domain analogical learning (Klenk & Forbus, 2007).

Our theory is that aggregation and set abstraction are useful representations for mapping *visual* analogs, and that re-representation using a shape-type hierarchy can address some cases of ontological mismatch, where similar ideas cannot be identified as such because they are represented with different symbols.

Model

In this section we will describe our theoretical models for the three kinds of visual abstraction in more detail.

Our representational architecture consists of propositions, each of which connects two symbols with a relation. For example

```
(butterdish looks-like rectangle)
```

connects the *butterdish* symbol to *rectangle* with a relation that the agent uses to align symbols with the same shape type. This uses the Covlan visual language (Davies & Goel, 2007).

Set and Aggregation Hierarchies

Sets are explicit visual objects with no shape. They have links to their members, and the members likewise have back-pointers to the sets. Sets can contain other sets as members. Sets get aligned to other sets through some similarity measure based on the shapes of their members, but the members themselves will not be mapped unless the agent has a specific reason to do so. The cognitive justification for this is introspective: we do not appear, for example, to align each paper clip in one pile to each binder clip in another pile unless there is reason to do so, even though we might see the two sets as similar.

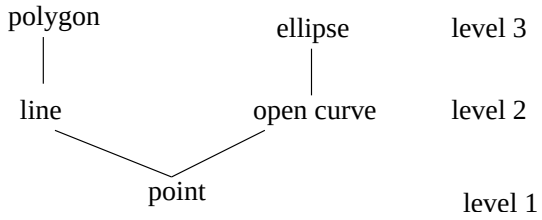

Figure 1: Component Aggregation Hierarchy

The same goes for aggregate objects. At the first pass of mapping, the aggregates are mapped to each other. In instances of conflict, the agent uses a measure of the shape similarity of the components to resolve it.

When there is a need to align the members or components, the entire mapping function can be recursively called on the sets or aggregates in question—that is, restart the mapping process as though the two aligned sets or aggregates were to the two images to be mapped.

The simplest kind of aggregates are visual elements aggregated together. Using knowledge of a shallow *aggregation hierarchy* (see figure 1) the agent can bring more domain knowledge to bear on the aggregate objects to align the components. When called upon to do align the components of simple visual elements, to resolve conflicts the most specific element in either aggregate object is decomposed into its aggregate parts. Then the identity-based mapping system tries again. This process repeats until all the align-able sub-elements are aligned.

Shape-Type Hierarchy

Each level of the shape-type hierarchy is associated with a level number. The higher the number, the less abstract the shape is. Abstraction is changing a shape to its more abstract form. For example, abstracting a *square* (level 8) means transforming it to a *rectangle* (level 7).

Process

Mapping is iterative. A *mapping* is a set of *maps*, which are alignments between a visual element in one analog to an element in the other.

1. Create maps of identical shape types, including aggregates and sets, ignoring the components of aggregates and the members of sets. If there is a conflict with mapping aggregates and sets, break them up into their constituents and see which are the most similar (a simple vector analysis of the contents).

2. If there are no more shapes to map in either the target or the source analog, recursively run this

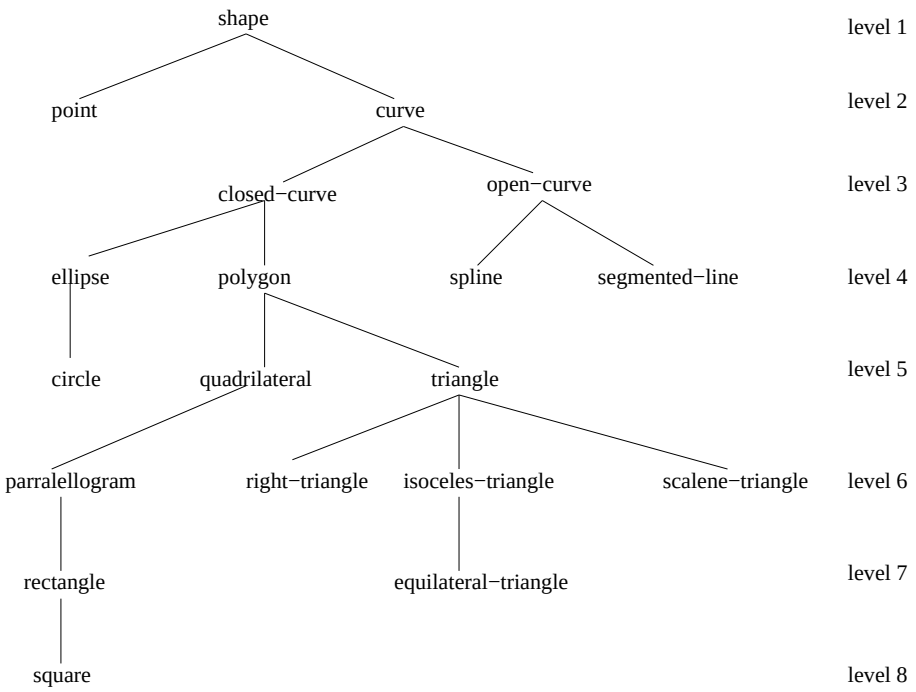

Figure 2: Shape Type Hierarchy

mapper on the members of sets and the components of aggregates, if any, then exit.

3. Abstract the highest numbered visual primitive in either analog one level. Go to step 1.

The visual elements need to be abstracted separately. If every element in the image is abstracted at once, matches will miss each other as they pass through levels of abstraction.

Implementation

We have some implemented the ideas above in a running computer program called *Thalassa*. Thalassa maps identical shapes and groups, and abstracts shapes with the shape-type hierarchy.

Thalassa has two basic components: a frame system and a "classical" problem solver. First, the "memory" of shape types and images with their elements and aggregate objects and such was built using a simple frame system. Relations of the sort `(building looks-like square)` lend themselves naturally to a frame-based representation, where the frame for `building` has a slot `looks-like` with the filler `square`. Likewise an image is a frame with a slot `contains-elements` whose filler is a list of elements (symbols naming visual element frames) in the system. Though the content of the frames in our actual implementation was sparse, the idea is that any extra information that might be useful to a larger problem-solving context could be added. For instance, surely people are aware of the location (qualitative or relative) within an image of a particular visual element, and the frame representation naturally allows one to add a slot `has-location` (or what have you). The only information actually used in the implementation was the `looks-like` slot for each visual element and the `has-size` slot (which took sizes like `small`, `medium`, and `big`).

Problem Solver

The second part of the implementation was the problem solver itself. Following the problem space hypothesis, and using the Classical Problem Solver (Forbus & DeKleer, 1993), we transformed the mapping problem into a search problem. One can think of one "state" of the search as the current set of maps—that is, the list of elements in each of the two analogs that have been mapped so far. An operator generating a new state in the search can do one of two things: map as many elements with the same shape as possible (generating a new state for each partial matching possible), or else pick one element to abstract.

The actual data structure representing a "state" in the search had the following elements:

Source The name of the source image frame

Target The name of the target image frame

Maps The list of maps gathered so far

Unmapped Source Elements A list of source elements that have not been mapped onto target

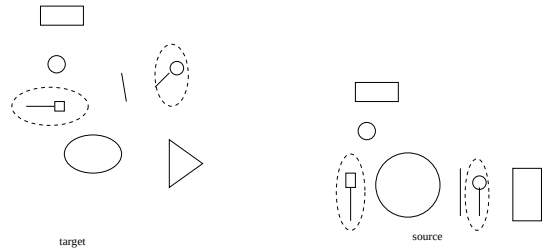

Figure 3: These two analogs represent a table before and after dinner. The fork and spoon are aggregate objects. Pictured are plates, butter dishes, forks, knives, spoons, and napkins.

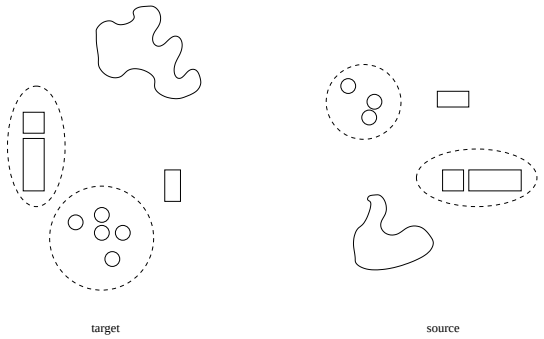

Figure 4: In the lot source there is a group representing a 18 wheeler (an aggregate of a square representing the cab and a rectangle representing the trailer). Also in the image is a set of garbage cans. Note there are a different number of cans in the analogs. The irregularly shaped object is a puddle, and the lone rectangle is a dumpster.

elements, initially set to all the source elements in the image.

Unmapped Target Elements A list of target elements that have not found source analogs, initially set to all the target elements in the image.

Needs Abstraction? A flag indicating that there are unmapped source and target elements that cannot be mapped without abstracting the shape types.

Abstractions A data structure that associates with each element (source and target) it's currently abstracted shape type. This is initially filled with the fillers from the `looks-like` slot, and as shapes are abstracted the contents slowly change.

The goal condition in this search is simply that either the `unmapped-source-elements` list or the `unmapped-target-elements` list becomes `nil`, in which case there are no more elements to match.

The next state operator `generate-new-mappings` simply checks the `needs-abstraction` flag, calling `abstract-one-element` if it is true, and `extend-mappings` otherwise. The `abstract-one-element` function is quite simple: it sorts the complete list of unmapped elements, choosing the one with the highest level arbitrarily (that is, if there are several, choosing the one at the front of the sorted list as an arbitrary choice) to abstract one level, if possible. It cannot abstract anything past `shape`, obviously (that being the top of the hierarchy), and so returns nothing if all of the shapes in the image are fully abstracted.

The `extend-mappings` function is much more complex. It has three parts: (1) generate all possible `maps-to` relations for each unmapped target element, separating them into whole mappings as it goes; (2) separate `maps-to` relations within each mapping that map onto the same source into separate whole mappings; and (3) generate a new state for each whole mapping.

The first part of this operation makes a list of all the elements that map to the given target (which is simply a list of all the unmapped sources that are identical under the `looks-like` relation and the current abstractions), and separates these into whole mappings, where one whole mapping has one `maps-to` relation for each target (there may be targets with no mapped sources, of course). It takes care to assemble all combinations when doing this.

The second part of the mapping looks within each mapping for two maps that map separate targets to the same source. If one is found, it is split into two mappings by removing one and then the other map from the mapping.

The third part simply takes each whole mapping, filters out those elements (sources and targets) which have been mapped from the unmapped elements lists, and generates a new state. A list of all new states is returned.

The search returns all mappings that it found. It's not clear to us that there is necessarily any cognitive plausibility in this decision, but for the sake of implementation we thought it best to have it return all mappings rather than choose one arbitrarily as the "best" mapping to return (one of the examples discussed below had several possible mappings).

One feature that has not been implemented in this version is recursing on aggregates and sets. Also, the system could often abstract in more than one way, and it's not clear that choosing one thing arbitrarily to abstract is the best decision; it seems wiser to have it abstract in all possible ways, generating new states (and hence new subtrees) in the search (space) for each one.

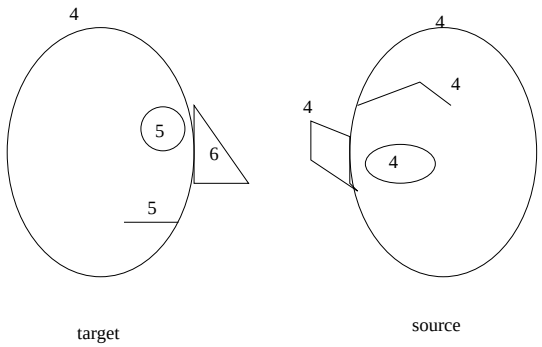

Figure 5: The faces example has similar faces reflected and rotated 180 degrees. Such transformations do not preserve many spatial relations, with certain exceptions such as containment and connections. The numbers represent the level of specificity of each shape in the shape-type hierarchy.

Test Examples

We ran this system on three test examples. The first, illustrated in Figure 3, was a table set up before and after a meal (not necessarily images from the same meal), where the elements are scattered about the table after the meal, and so structure would probably be uninformative. However, a fork looks like a for, a plate looks like a plate, and so on. In fact, the system found four mappings: the fork could map to the fork or the spoon (and vice versa), and the napkin could map to the napkin or the butter dish (and vice versa). Everything else mapped to the element of the same name, thus giving four mappings.

The second, illustrated in figure 4, was supposed to be an overhead view of a parking lot, with a dumpster and a group of trash cans and a truck aggregate (cab and trailer) and a puddle all in different positions. this had only one mapping.

The third example, illustrated in figure 5, mapped a face to another face reflected and inverted. Again, only one mapping was found.

Discussion

We are theorizing about solutions to the cognitive problem known as the ontological mismatch problem: When two ideas that should be thought of as similar are not because they are represented with different symbols. This problem manifests itself in mapping because the labels for objects and relations often do not match exactly. For example, mapping *orbits* with *revolves-around*.

Our solution uses shape-type abstraction, which is a content account of the visual domain. EMMA (Ramscar & Yarlett, 2003) uses a content account as well to solve ontological mismatching for analogy. The knowledge EMMA uses is the correlation of word proximity in text (Latent Semantic Analysis, LSA). Each word in LSA correlates with each other word. When trying to map, different words with a correlation above a certain threshold are considered equivalent, and can be mapped.

The Structure-Behavior-Function knowledge representation language (Goel et al., 1997) offers another means for resolving ontological mismatches. SBF language representations of systems include functional descriptions each device component.

Forbus et al. (1998, p.246) and Hummel and Holyoak (1997) both suggest that ontological mismatches can be resolved through re-representation using abstraction (e.g. *lift* and *push* can abstract to *move*. This idea also suggests a superclass hierarchy, but to our knowledge neither research group has implemented this.

Other implemented mappers rely on the identicality of symbols (either of the mapped concepts or the relations between them) and on a canonicalized representation to avoid ontological mismatches.

Ours is a content based account that uses re-representation using domain knowledge of visual objects.

Conclusion

In this paper we have described a method based on domain knowledge for analogical mapping of visual representations. Specifically, we focused on grouping and shape-abstraction. The one-to-one mapping constraint is maintained for our system, but sets and aggregates are treated as object to be mapped. Our ideas are implemented into a running computer program called Thalassa. Future versions of Thalassa will be able to recursively map set members and aggregate components.

We conjecture that these content-based mapping strategies will prove superior to structure-mapping in cases when the analogs share similarly-shaped components but where the spatial arrangement of the objects is disordered. A full, cognitively plausible model of visual mapping will need to include both structure and object mapping. Future research will test these claims through computational comparison with structure-based mapping engines on several examples.

References

Davies, J. & Goel, A. K. (2007). Transfer of Problem-Solving Strategy Using Covlan. Journal of Visual Languages and Computing: 18, 149–164.

Davies, J., Goel, A. K. & Nersessian, N. J. (2003). Visual Re-Representation in Creative Analogies. In A. Cardoso & J. Gero (Eds.) *The Third Workshop on Creative Systems.* International Joint Conference on Artificial Intelligence, 1–12.

Falkenhainer, B. (1988). *Learning from Physical Analogies.* Technical Report No. UIUCDCS-

R-88-1479, University of Illinios at Urbana-Champaign. (Ph.D. Thesis)

Forbus, K., & De Kleer, J. (1993). *Building Problem Solvers.* MIT Press.

Forbus, K., Gentner, D., Markman, A. B., & Ferguson, R. W. (1998) Analogy just looks like high level perception. Why a domain-general approach to analogical mapping is right. *Journal of Experimental and Theoretical Artificial Intelligence*, 10, 231–257.

Gentner, D. (1983). Structure-mapping: A theoretical framework for analogy. *Cognitive Science*, 7, pp 155-170.

Goel, A., Bhatta, S. & Stroulia, E. (1997) Kritik: An Early Case-Based Design System. In Maher, M. and Pu, P. (Eds.) *Issues and Applications of Case-Based Reasoning in Design*, Mahwah, NJ: Erlbaum, pages 87–132.

Hummel, J. E., & Holyoak, K. J. (1997). Distributed representations of structure: A theory of analogical access and mapping. *Psychological Review*, 104, 427–466.

Klenk, M. & Forbus, K. (2007). Cross domain analogies for learning domain theories. In A. Schwering et al. (Eds.) *Analogies: Integrating Multiple Cognitive Abilities*, Volume 5-2007. Publication of the Institute of Cognitive Science, University of Osnabruck.

Ramscar, M. & Yarlett, D. (2003). Semantic grounding in models of analogy: an environmental approach. *Cognitive Science* 27:1. 41–72.

Output

```
CL-USER(33): (find-mappings 's-face-simage
't-face-simage) ; Fast loading
/net/hc283/yaner/work/current/7613/bps/proj/memory.fasl
(((S-FACE-HEAD MAPS-TO T-FACE-HEAD)
(S-FACE-MOUTH MAPS-TO T-FACE-MOUTH)
(S-FACE-EYE MAPS-TO T-FACE-EYE)
(S-FACE-NOSE MAPS-TO T-FACE-NOSE)))
CL-USER(34): (find-mappings 's-lot-simage
't-lot-simage) ; Fast loading
/net/hc283/yaner/work/current/7613/bps/proj/memory.fasl
(((S-PUDDLE MAPS-TO T-PUDDLE)
(S-TRUCK-AGGREGATE MAPS-TO
T-TRUCK-AGGREGATE) (S-DUMPSTER
MAPS-TO T-DUMPSTER) (S-CANS
MAPS-TO T-CANS))) CL-USER(35):
(find-mappings 's-table-simage
't-table-simage) ; Fast loading
/net/hc283/yaner/work/current/7613/bps/proj/memory.fasl
(((S-NAPKIN MAPS-TO T-BUTTERDISH) (S-GLASS
MAPS-TO T-GLASS) (S-FORK-AGGREGATE
MAPS-TO T-SPOON-AGGREGATE) (S-KNIFE
MAPS-TO T-KNIFE) (S-SPOON-AGGREGATE
MAPS-TO T-FORK-AGGREGATE) (S-PLATE
MAPS-TO T-PLATE) (S-BUTTERDISH MAPS-TO
T-NAPKIN)) ((S-NAPKIN MAPS-TO T-BUTTERDISH)
(S-GLASS MAPS-TO T-GLASS) (S-FORK-AGGREGATE
MAPS-TO T-FORK-AGGREGATE) (S-KNIFE MAPS-TO
T-KNIFE) (S-SPOON-AGGREGATE MAPS-TO
T-SPOON-AGGREGATE) (S-PLATE MAPS-TO
T-PLATE) (S-BUTTERDISH MAPS-TO T-NAPKIN))
((S-BUTTERDISH MAPS-TO T-BUTTERDISH)
(S-GLASS MAPS-TO T-GLASS) (S-FORK-AGGREGATE
MAPS-TO T-SPOON-AGGREGATE) (S-KNIFE
MAPS-TO T-KNIFE) (S-SPOON-AGGREGATE
MAPS-TO T-FORK-AGGREGATE) (S-PLATE MAPS-TO
T-PLATE) (S-NAPKIN MAPS-TO T-NAPKIN))
((S-BUTTERDISH MAPS-TO T-BUTTERDISH)
(S-GLASS MAPS-TO T-GLASS) (S-FORK-AGGREGATE
MAPS-TO T-FORK-AGGREGATE) (S-KNIFE MAPS-TO
T-KNIFE) (S-SPOON-AGGREGATE MAPS-TO
T-SPOON-AGGREGATE) (S-PLATE MAPS-TO
T-PLATE) (S-NAPKIN MAPS-TO T-NAPKIN)))
CL-USER(36): (dribble)
```