Architecture for analogical reasoning in a Bongard-problem solver

Lanfranco Muzi

Abstract. This thesis describes an architecture implementing the "Analogy-Making as Perception" model of Mitchell and Hofstadter (1990) for solving Bongard problems. This particular class of visual pattern-recognition problems was introduced in 1970 by the Russian intelligence theorist Mikhail M. Bongard and still represents a formidable challenge for AI research. In general, the complete solution of a Bongard problem starting from raw visual input (e.g., a digital image) would require the integration of low-level visual processing with some model of perceptual organization. This system provides a perceptual-organization engine that performs high-level reasoning on visual objects represented by geometric figures. In this system the solution to a Bongard problem emerges as a result of the interaction between an adaptive network of concepts and the work of a swarm of simple software agents. The concept network represents the program's prior knowledge, whereas the agents explore the input and automatically build a set of relevant features and the structures representing their mutual relationships. The behavior of the agents is partially stochastic and partially driven by the activation patterns emerging in the concept network as the system's "understanding" of the problem proceeds. This system provides an architectural basis for a future complete Bongard-problem solver. Besides introducing Bongard problems and describing my system, I also describe the theoretical foundations (both in the field of cognitive science and in the field of complex systems) upon which the architecture is based.

Table of Contents

1. Introduction

The field of computer vision studies how a computer can extract descriptions of the world from pictures or sequences of pictures. The set of its applications is continually expanding and includes fields such as robot navigation, industrial and military inspection, human-computer interaction, computer graphics, medical image analysis, and image retrieval in digital databases. Although no comprehensive "theory of vision" has been developed so far, a commonly accepted approach subdivides the set of processes involved in computer vision into two broad classes (see for instance Marr's pioneering work [1]): *Low-* and *high-level* vision. Low-level tasks include filtering out sensor noise and detecting visual primitives such as edges, corners, and texture. These operations are generally characterized by a *local* nature: Each pixel is processed together with a restricted number of neighboring pixels. For instance, Canny's algorithm for edge detection [2, 3] is based on the brightness of pixels and the orientation of each partial edge with respect to neighboring edges. These algorithms are called *bottom-up*, because they yield results solely by processing the raw pixel data.

Segmentation is the task of identifying different regions within an image by subdividing the pixels in groups according to some criterion, such as distinguishing foreground from background, or different objects in the picture. Segmentation is at the border between low- and high-level vision.

The field of high-level vision includes complex tasks leading to the final goal of *image understanding*, such as shape characterization, object recognition, and extraction of three-dimensional information. These kinds of tasks often require the machine to recognize some pattern and find out whether that pattern is also present in new images. At this level, purely bottom-up approaches become error-prone [3]. To perform acceptably, computer vision systems need some knowledge of what kind of patterns are likely to be encountered in the images. The development of algorithms and techniques for defining and extracting the features that identify a pattern is a very active research field, while machine-learning techniques are generally employed to train the system and measure similarity between patterns [3]. Examples of image-understanding applications include criminal-behavior detection on the basis of movement pattern [4], handwriting recognition [5], and image search in medical databases through content-based image retrieval [6, 7].

While the importance of such applications has led to the development of many dedicated systems, general image understanding has proven to be an extremely difficult task for machines. Existing systems typically do not generalize to problems outside the specific task for which they were engineered. The lack of flexibility exhibited by computer-vision programs contrasts sharply with the versatility and robustness of their biological counterparts. The ability of biological beings to organize the stream of elementary bits of information provided by the senses (e.g., a retinal image) into a set of larger units (e.g., different objects in a visual scene) and their interrelations is called *perceptual organization*.

At the basis of this work is the idea that the superior performance of biological vision systems is due, in part, to the interaction and overlapping of low- and high-level processes, a characteristic that is often not captured by the less versatile man-made systems, which often model them as a pipelined series of information-processing stages. Churchland, Ramachandran, and Sejnowski [8] propose an alternative theory of *interactive vision*, according to which the versatility of biological vision emerges from the continual, parallel interaction of processes spanning the whole range between low- and high-level vision. A pipelined approach lacks this integration of low-level processing and high-level cognitive activities. In the theory of interactive vision, the perceptual organization of visual input *emerges* from interaction between low-level processing modalities, other sensory input, and high-level knowledge. The architecture presented in this thesis is closely aligned with this interactive-vision philosophy and other cognitive theories.

In the following sections, I introduce *Bongard problems* and explain why they represent a useful domain for image-understanding systems. I argue that the abilities required in this task are fundamental in real-world applications that are hard for current automated systems. I then describe how biological cognitive systems tackle these tasks, and introduce *analogy-making* as the cognitive process at the basis of their superior performance (with respect to computers). I describe how cognitive theoreticians have formalized the theory of analogy-making and summarize some computer models implementing different theories, highlighting the differences between the approaches they advocate. I then contrast the classical approach to computer vision and the interactive-vision approach. I highlight interesting similarities between interactive vision and the cognitive theory of *high-level perception* and I add recent findings in the field of complex adaptive systems as further theoretical foundation for the approach I suggest. Finally, I introduce my architecture and cover its functionality.

2. Bongard problems

I propose capturing some of the mechanisms of interactive vision and other cognitive theories in a computer program, and believe that Bongard problems (BP) represent an excellent class of idealized problems for the test and validation of the architecture. The first 100 Bongard problems were introduced by the Russian intelligence theorist Mikhail M. Bongard in his 1970 book on pattern recognition [9], and still represent a formidable challenge for AI research. Each problem consists of two *sets* of figures (see Figure 1); the pattern recognizer must find some concise description that distinguishes all the figures in one set (six *frames* on the left) from those in the other set (six frames on the right). I add here a remark that in a correctly designed Bongard problem, there can be no ambiguity on what this distinctive description is: *The solution is always unique*. Two sample problems, not included in Bongard's collection, are shown in Figure 1. Although the examples in the figure appear quite simple, the class of problems that can be conceived in a "Bongard fashion" is actually very open-ended, and much harder problems can be found both in Bongard's original collection and in the literature [10]. For instance, BP#64 and BP#20 in Figure 2 [9] have generally been considered much harder by some subjects polled than those in Figure 1 (harder examples are shown in later sections).

As Figure 1 suggests, the solution of Bongard problems by a machine, starting from the raw bitmap images, will in general require both skills that are typical of low-level image processing (e.g. edge detection, texture identification, segmentation) and high-level vision skills, such as shape identification (with possible occlusion), absolute and relative positioning, recognition of relationships among objects, and grouping of objects.

Recognizing a set of circles as a triangle (P2, group a, upper left), or a set of parallel straight lines as a circle (P1, group a, lower right) is an exercise of high-level vision in which the cognitive process called *analogy-making* plays a major role [11, 12]. Analogy-making is the process through which one can perceive two different situations as being similar at some abstract level. Defined in such general terms, "analogy-making" includes much more than problems of the form "A is to B as C is to what?". In fact most, if not all, cognitive scientists agree that analogy-making plays a fundamental role in general cognition [11, 13 – 16]. I now use a couple of informal examples to introduce the distinguishing traits of this cognitive process. For instance, when studying the second law of thermodynamics ("heat naturally flows only from a body at higher temperature to a body at lower temperature"), an analogy with the natural flow of water from a higher to a lower location appears to help greatly in understanding the nature of the phenomenon. Human beings have eased the process of understanding a novel situation by mapping something they know (the natural flow of water) to what they perceive as new and unknown. The process often involves mappings between concepts that are generally semantically distant, but turn out to be related in the particular context of interest.

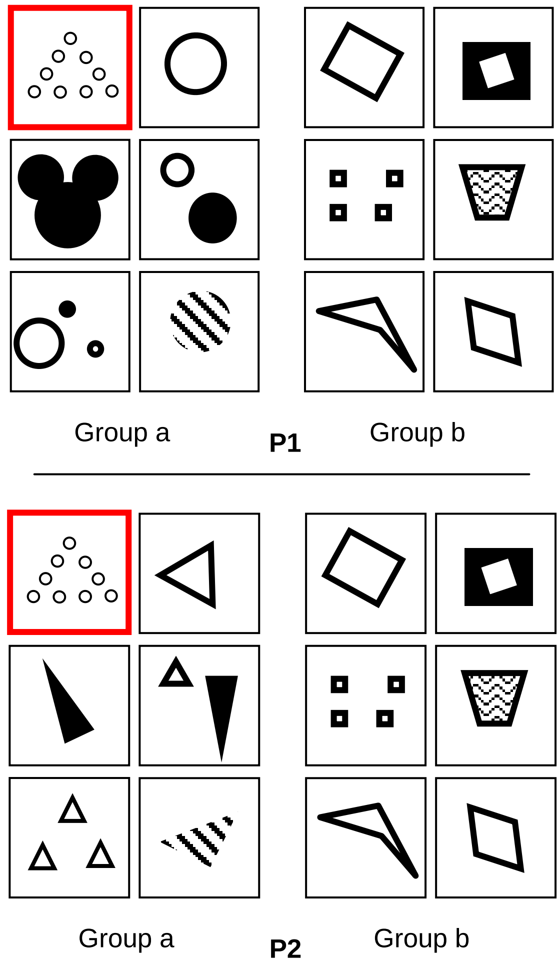

Figure 1. Two simple Bongard problems. Each problem consists of two sets of figures. The pattern recognizer must find some concise description that distinguishes all the figures in one set (six frames on the left) from those in the other set (six frames on the right). The answer to P1 is “circles vs. quadrilaterals”, and to P2, “triangles vs. quadrilaterals”. The frame in the top left corner of both problems is the same, as are the six frames on the right side.

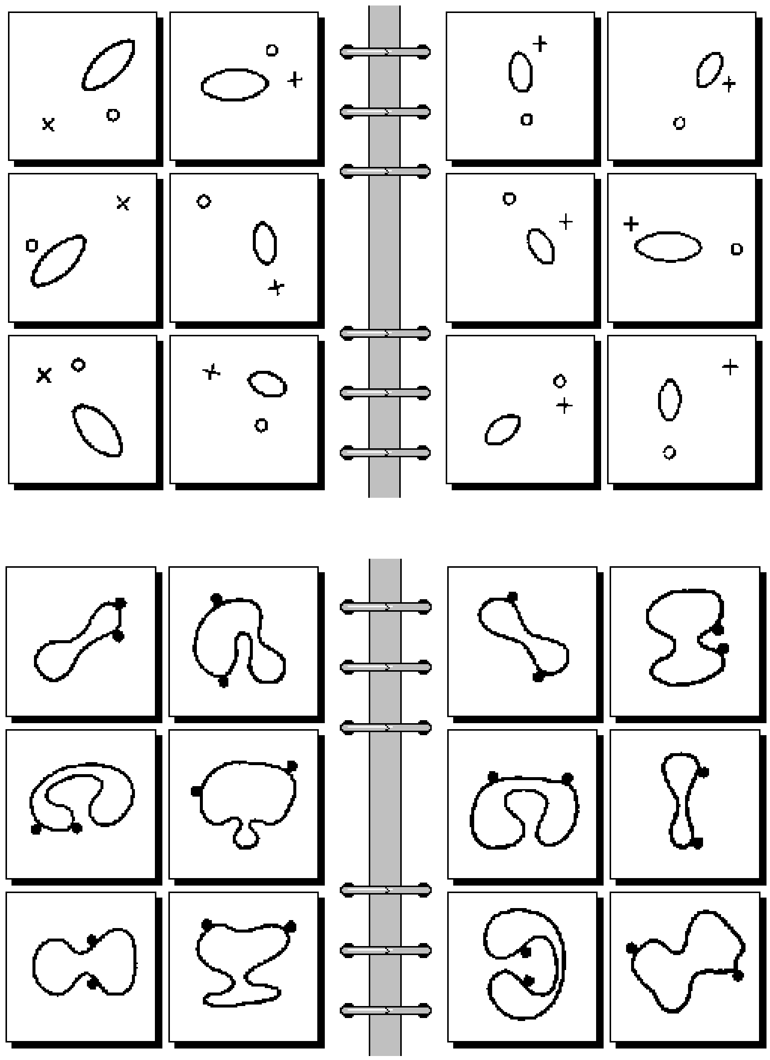

Figure 2. BP#64 (top) and BP#20 (bottom) from Bongard's original collection [9] (images from H. Foundalis' web site [10]). These are ranked as fairly difficult by human solvers.

Hofstadter [17], Linhares [18], and Foundalis [19] argue that solving Bongard problems requires the ability to *perceive abstract similarity depending on the specific context*. In Bongard problems, the solver must see the six frames in each set as similar, but what exactly this similarity consists of depends on the specific problem under consideration. In Figure 1, for instance, the upper-left-corner frames of P1 and P2 and the six frames on the right in each problem are exactly the same. Yet, the solutions to the two problems (P1: “circles versus quadrilaterals”, P2: “triangles versus quadrilaterals”) are different. To solve problem P1 one needs to map the shape (“circle”) of the objects in the upper left frame of P1 to that of all the other objects in the six frames on the left. To solve problem P2, one needs to consider the shape (“triangle”) outlined by the centers of the circles in the upper left frame, and map it to the shape of the other objects on the left side. One must map an abstract notion of triangle to a set of circles, changing what they see in the upper left corner under the pressure of the context provided by the rest of the Bongard problem.

The difficulty of implementing analogy-making makes Bongard problems very hard for an automated solver, but I believe that humans employ this cognitive process extensively to carry out tasks with great potential impact on computer science applications, such as image understanding. For instance, a human viewer can easily recognize the subject of a set of pictures as “dogs”, regardless of breed, color, pose, and whether the picture is a photograph or a drawing. Humans have an impressive ability to (a) identify sets of abstract features that define a concept of “dog”, (b) organize a matrix of colored pixels into one or more relevant objects, and (c) build consistent mappings (i.e., an analogy) between these objects and those abstract features. A computer program aimed at automating image understanding should carry out these same tasks. Bongard problems, in a much simpler context, require the same abilities from a solver. This is why I believe that investigating the application of computer models of analogy-making to Bongard problems is a first step towards a better understanding of how a general image-understanding system should be designed.

In summary, Bongard problems capture in an idealized domain the essence of perceiving abstract similarity. A solver for these problems should therefore implement a model of analogy-making and use it to find the solution. The idealized domain of Bongard problems, of course, leaves out many aspects of the image-understanding problem, such as color, extraction of three-dimensional information, complex object recognition and movement. However, I believe that the mechanisms that one needs to capture in a Bongard-problem solver have *general* applicability to real-world image-understanding problems.

3. Theories of analogy-making

The distinguishing trait of this work is the incorporation of analogy-making into a visual-pattern recognition program. Different theories exist on how to explain the mechanics of analogy-making, and the choice of a particular theory imposes a number of constraints on the design of a computer model of this process. Kokinov and Petrov [20] provide a detailed analysis of these constraints, whereas French [21] has written a good review of all the major analogy-making computer models. In this discussion, it will suffice to distinguish two levels at which cognitive phenomena take place [14]: *perception* (i.e., processing of the sensory input) and *conceptualization* (i.e., extraction of a "meaning" from the results of perception). The debate is still open on whether analogy-making is best modeled as a strictly conceptual manipulation of symbols resulting from perception, or if it should instead be modeled as being integrated with perception [11, 14–16, 22–27].

In artificial intelligence, it is common to treat symbol manipulation as a self-contained stage that follows the preprocessing of sensory input in a serial fashion. Instances of this school of thought are the classical approach to computer vision briefly described above, Newell's physical symbol systems [22], and, in the field of analogy-making, Gentner's structure-mapping theory [28]. I will refer to this approach as the *modular approach*.

An alternative approach suggests that perception and conceptualization share the same mechanics and take place simultaneously, significantly influencing each other. Examples of this school of thought are Barsalou's conceptual symbol systems [27], Churchland, Ramachandran, and Sejnowski's theory of interactive vision [8], and Hofstadter's theory of high-level perception [11]. I will refer to this approach as the *perceptual approach*.

4. Models of Analogy-making

4.1 The Structure-Mapping Engine

An example of computerized analogy-maker based on the modular approach is the Structure Mapping Engine (SME), developed by Falkenhainer, Forbus and Gentner [29] and based on Gentner's structure-mapping theory [28]. The program receives the description of two situations, the *source* and the *target*. For instance, in the earlier example of thermodynamics, the source would be the flow of water between two communicating containers where the fluid is at different levels, and the target would be heat transfer between two objects at different temperatures. SME produces an analogy by mapping corresponding elements in the two descriptions (e.g., water level maps to temperature, water maps to heat). The input descriptions are in the form of predicate-calculus formulas[1] prepared by a human or another program. These formulas provide an explicit representation of the objects involved in the problem, their properties, and the relations between them. According to Gentner's theory, analogy-making consists of detecting similarities in the structure of these relations. When structural similarities in two relations exist (one in the source domain and one in the target domain), and if they satisfy some constraints that are also defined in the theory, the objects they involve in the source domain are mapped to the corresponding objects in the target domain. This approach is clearly modular, because it separates in principle the building of a representation of the input problem (which is not considered part of the analogy-making process) from the mapping stage. Forbus, Gentner, Markman and Ferguson [26] report that several cognitive-simulation programs employ SME as a component module.

Although I do not question the correct implementation of the structure-mapping theory in SME and its cognitive relevance, I believe that this specific approach to analogy-making modeling is not suitable to automate a Bongard-problem solver. This claim stems from the discussion in

---

1 An example reported by Falkenhainer, Forbus and Gentner [29] is: `greater(size(A), size(B))`, indicating that we are considering objects A and B, and that A is greater in size than B.

Section 2: it is impossible, in this kind of problem, to describe what is relevant to the solution, without actually getting into the process of solving the problem. What information should be included in a predicate-logic description of the upper left corner frame of P1 in Figure 1? What information should be included in the description for the same frame in P2? In P1, the shape of the objects in the frame is all that matters, whereas in P2 it is the position of the centers of the circles that is relevant to the solution. And (stealing an example from Linhares [13]) how would one describe the objects in the problems of Figure 3?

The class of Bongard problems is so open-ended that it is impossible to build a complete description of a problem *a priori*, including *all* the information that could possibly be relevant in the search for the solution. But the idea of providing the solver with a complete description of the problem must also be rejected in principle [18], because discovering relevant information is an important part of any image-understanding task, and should therefore not be left out of the model and delegated to the human user or a separate automated module. A complete BP solver should be able to build its own representation of the problem and, in the process, make decisions as to what is relevant. It should analyze the images, collect information on each frame, and assess how well it fits in the *context* of the entire problem.

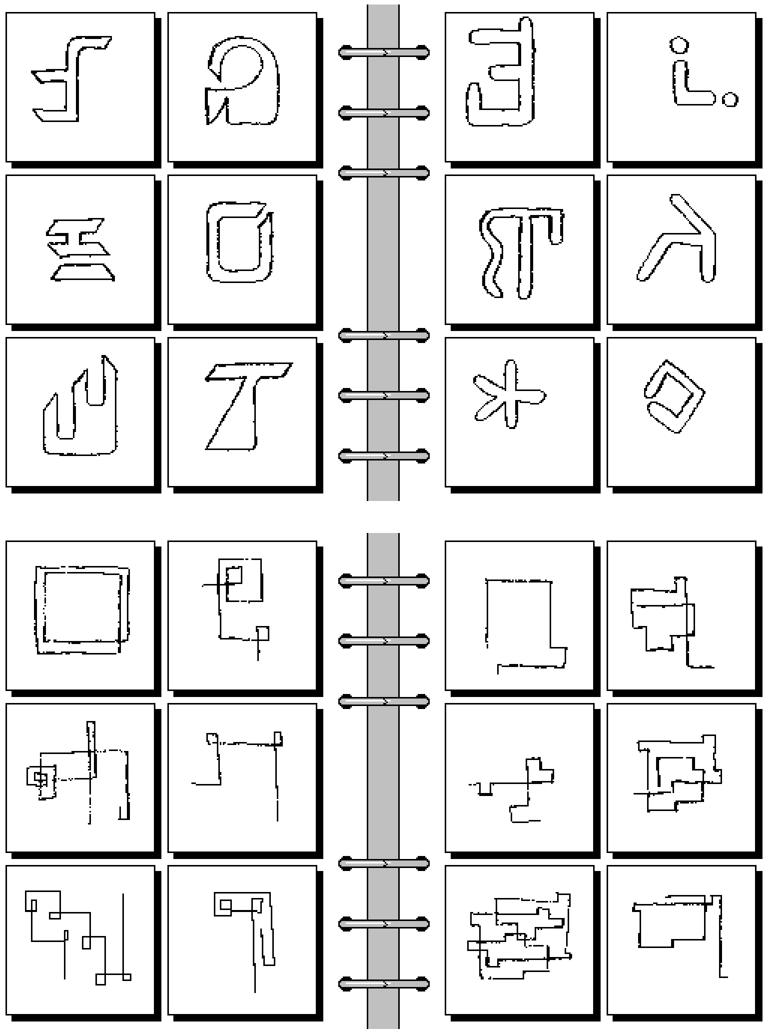

Figure 3. Problems H54 (top) and H20 (bottom) from H. Foundalis' web site [10]. Author: D. Hofstadter.

4.2 Copycat

I provide here a brief description of Mitchell and Hofstadter's Copycat [30, 31], as an example of an analogy-making program based on the perceptual approach. Copycat has a "blackboard-like" architecture that was partly inspired by the Hearsay-II speech-understanding system [32]. I provide a concise description here, while Section 6 provides theoretical motivations and explains in detail how a system of this kind works.

Copycat works in a microdomain of letter-string puzzles. It receives three strings defining a problem and must produce a solution. An instance of this kind of problem is: "If **abc** changes to **abd**, how should **kji** change?"

String **abc** is called the *initial* string, **abd** is the *modified* string and **kji** is the *target* string. The user does not provide the program with any clues on the problem, except for the order in which the letters appear in the strings. Copycat must analyze the initial and the modified strings and build its own explanation for the change necessary to derive the latter from the former. It must also understand how to see the target string as "analogous" to the initial string. When the program has formed this analogy, it modifies the target string "in the same way" as the user modified the initial string to produce the modified string. This modification is Copycat's answer to the puzzle.

The biggest part of Copycat's work is finding the *structure* of the problem. For instance, in the puzzle above, the program needs to understand that **a**, **b** and **c** in the initial string correspond to **a**, **b** and ***d***, respectively in the modified string. It must also understand that the group **abc** has the key feature of having its letters in increasing alphabetical order. Both *correspondences* and *groups* are examples of structural elements of the problem that Copycat can discover.

Copycat's behavior is partly nondeterministic: Every decision is made probabilistically, and the probability distribution is a function of a parameter called temperature, which measures the amount of perceptual organization reached by the problem at each step. As the program's "understanding" of the problem progresses, the temperature lowers and the program becomes more focused on exploiting the most promising hypotheses.

The architecture implements some of the fundamental mechanisms of thought in Hofstadter's theory of high-level perception [11]: The *bottom-up* influence of perception on conceptual processes; the *top-down* pressure that is born when concepts begin to get activated in the mind; the continual *interaction* of these processes. This general theoretical framework for human cognition shows important similarities to the above mentioned theory of interactive vision [8] and to some findings in the field of complex adaptive systems [33]. Its relevance to my system is explained in detail in Section 6, where I also describe the architecture of the system developed for this thesis. I add here that Copycat's architecture does not include some aspects of cognition. In particular, the model does not include long-term memory, memory retrieval and learning. Marshall later extended the architecture to explore these processes [34]. Elements of Copycat's architecture have been included into artificial-intelligence systems also by researchers outside Hofstadter's group (see, for instance, Dor's work in sound perception [35] and Bogner, Ramamurthy and Franklin's work on "conscious", interactive computer agents [36]).

5. Current status of Bongard-problem solvers

There have been relatively few attempts at building an automated solver for Bongard problems, partly because the field of artificial intelligence has gradually shifted its focus towards more practical (not necessarily more difficult), narrower “real-world” problems, and partly, I believe, because Bongard problems still represent an extremely hard challenge for a AI.

In 1993, Saito and Nakano [37] presented a paper on RF4, a concept-learning and adaptive-search architecture and reported that it had been applied successfully to a variety of search problems, including a set of 41 problems from Bongard’s collection. The system requires input not as images but in the form of human-compiled, first-order logic formulas, and therefore the same criticisms that were raised in Section 4.1 for SME apply to RF4.

More recently, Foundalis [19] has developed Phaeaco, an architecture derived from Copycat and its descendants, that implements features of high-level perception not present in Copycat. The architecture has been specialized in particular for solving Bongard problems. The system includes several components: *short-term memory*, *long-term memory*, a *pattern-matching* mechanism, a *clustering* mechanism (to group objects found in the frames), and a *categorization* mechanism (which produces the solution).

Unlike Copycat, the system is explicitly organized in two levels. The *retinal* level begins the work by performing low-level processing of the visual input. The *cognitive* level starts as soon as some entities have been perceived by retinal activity. This second level builds representations, in the form of graphs, of the structures it identifies in its input. The input is parsed several times and, given the probabilistic nature of the choices the system makes, the structures and their representations can differ each time. The different graphs (“views”) are then condensed into a *visual pattern*, *i.e.*, a new graph whose nodes and links are associated with statistics reflecting how often two nodes have been seen linked in the set of views examined.

The detailed functioning of Phaeaco is beyond the scope of this thesis. However, I want to highlight here some features that differ from the system described in this thesis. Phaeaco does *not* work in parallel on the entire input, constantly seeking correspondences among the structures found at each step. The approach is more "staged": To build the patterns, the system performs a deterministic graph matching, and when the global patterns of the two sides of the problem have been built, the solution is again the result of this deterministic activity. My proposed system does not separately implement probabilistic and deterministic stages of processing. Borrowing mechanisms from natural complex systems, the program shifts from probabilistic to deterministic behavior, as self-organization emerges from the work of many local, simple agents with no centralized control. Section 6 explains the motivation for this architecture and why this approach was adopted for this work.

6. System description

6.1 Theoretical foundations of this work: Cognition as a complex adaptive system

I mentioned before that Copycat implements some principles of Hofstadter's theory of *high-level perception* [11, 14]. The main argument of high-level perception is that biological cognitive systems do not achieve perceptual organization through a pipeline of deterministic processes: Their flexibility and robustness emerge from the parallel execution of a high number of subconscious elementary activities and conscious processes driven by high-level knowledge [11].

In a computerized system, this scheme can be seen as the interaction between low-level input processing and a body of higher-level prior knowledge. Systems of this kind have been developed or proposed for other specific tasks, such as robot navigation [25], image interpretation [38], and human audition [39]. Copycat's architecture in particular was inspired by the Hearsay-II speech-understanding system [32], and represents, to my knowledge, the first example of a working system that successfully addressed these general cognitive issues in the framework of analogy-making. Churchland, Ramachandran and Sejnowsky's theory of *interactive vision* [8] and Barsalou's *perceptual symbol systems* [27] provide support for this view from the field of cognitive science.

Although Hofstadter began to develop his theory [17] when complex adaptive systems was not an established research field, successive studies in this field provided further theoretical support to his ideas and some insight into why the strategy implemented in Copycat is successful. As Mitchell points out [33, 40], researchers have gained significant insight by studying complex biological systems such as the vertebrate immune system, ant colonies, and cellular metabolism as adaptive information-processing networks. From this viewpoint, the collective behavior of these highly decentralized systems shows adaptation, learning, memory, pattern recognition, and collective global control. Mitchell has identified four basic principles that seem to account at least in part for these abilities. I now introduce these principles in general terms. In the following paragraphs, when I mention "agents" in a "system", the reader can think of lymphocytes in the immune system, or individual ants in an ant colony. Afterwards, I present the same principles in the framework of a system aimed at solving Bongard problems.

1. **Collective behavior is the macroscopic effect of actions performed at a very fine-grained level**. Biological complex adaptive systems are often characterized by a large number of agents carrying out tasks that are very simple, compared to the behavior of the entire system. An advantage of this structure is that agents simultaneously make many small steps along many different paths towards the solution of the problem the system must cope with. This allows the system to explore different possible strategies *in parallel*, and to dynamically adjust the resources (agents) allocated to these different explorations. The outcome of each agent's action can be used to guide the behavior of the others: if a significant number of agents has attempted to perform a certain task, without any benefit for the entire system, other agents will be less likely to attempt the same task in the future. Meanwhile, other agents have been attempting other tasks, providing the system with viable alternatives. The system can readily begin to withdraw agents from the unsuccessful tasks and allocate them to more promising ones. The main advantages of this fine-grained architecture are *efficient resource allocation* and *robustness*: By constantly exploring many different paths the system avoids *exclusive commitment* to any *specific* strategy, while more resources are devoted to the more promising ones.
2. **Randomness and probability play an essential role in the functioning of the system.** Instead of having deterministic rules that direct agents towards predetermined goals, there is a certain amount of randomness in the behavior of each agent. This way the system carries out a broad-spectrum search in a possibly immense space, with a comparatively small number of

agents. This is particularly important when the system has "little a priori knowledge about what will be encountered" [33] — we will see that this is the case for a Bongard-problem solver.

3. **Global control is decentralized, and uses information encoded in the form of statistics.** Decentralized systems do not have any recognizable component dedicated to supervising their collective behavior. Every agent can interact only *locally*, with neighboring agents. Each interaction provides clues to the agent as to what other agents have been trying to do and what the outcome was. When the clues consistently indicating success along a certain path crosses a threshold, the agent is likely to "join the stream" and attempt to follow that path too.
4. **Top-down and bottom-up processes continually overlap and influence one other.** Bottom-up processes are those activities that process raw data, providing input to the system without using higher-level prior knowledge. Top-down processes use the input to build *expectations* that drive future actions by the system. When facing a novel situation, at first bottom-up processes must prevail, because the system needs to explore the input, in order to discover relevant information. Immediately agents begin to exchange information on the outcome of the bottom-up exploratory activity, and this begins to influence their future actions (see point 3 above). When evidence suggests that a given course of action is leading the system towards achieving its global goal, top-down pressure must build up to allocate more and more agents to that search path. The ability to balance dynamically bottom-up and top-down processes seems to be a distinguishing trait of adaptive intelligent systems.

The basics of Hofstadter's theory of high-level perception [11] introduced above bear some relevant similarities to these principles. The presumption at the basis of this work is that these similarities indicate that *information processing in human cognition could follow these same principles*. I believe that a computer-vision program implementing a model of image-understanding based on high-level perception should therefore be imbued with the flexibility and robustness necessary to meet, at least in part, Bongard's challenge. The study is of interest because the architecture, once developed, should provide insight into new mechanisms useful for general problems of image understanding.

6.2 Design constraints

The goal of this thesis is to build a system that implements the principles of the "Analogy-Making as Perception" model of Mitchell and Hofstadter (1990) into an architecture suitable for solving Bongard problems. It appears natural to choose Copycat's architecture as a starting point for the design of my program because:

- It implements a model of analogy-making;
- The model generally complies with the principles listed in Section 6.1, which are claimed to be at the basis of the robustness and flexibility of decentralized adaptive systems.

The following general design requirements can be derived from the preceding discussion:

1. Actions in the system shall be performed by simple, independent agents.
2. Non-determinism shall play a role in the decisions that the program must make.
3. No central supervisor shall be included in the program.

4. The system must be able to dynamically balance bottom-up and top-down activities.
5. The program must receive as input an incomplete, at least partially unstructured representation of the problem, and must show the ability to build structure relevant to finding the solution to the problem.
6. The program must, of course, produce the correct answer to a set of test problems, designed to check the sanity of the system.

6.3 System overview

Although several programs have been developed in Hofstadter's group to address specific "microworld" problems [11, 19, 30, 34, 41], no general guidelines exist for assembling or tuning such a probabilistic system towards convergence. I try here to abstract the *general* traits of these systems by describing the main components of the architecture: The *codelets*, the w*orkspace*, the c*oncept network*, and the c*oderack*.

Codelets are small, independent pieces of code, each able to build, destroy or change the state of some object in the workspace. The *workspace* is a collection of objects that are visible to all codelets. In our case, the workspace initially contains two sets of polygons, but it will later include any structural elements that have been found and built by the codelets. The *coderack* is a list of codelets that are waiting to run. The program advances in steps: at every step, a codelet is chosen in the coderack and run. The choice is made probabilistically, with the probability associated to each codelet depending on its *urgency*. The urgency is assigned to a codelet the moment it is posted on the coderack; this could be done by another codelet or by a node in the concept network (more on this later). The higher a codelet's urgency, the higher the probability that it will be picked for running in the future. Initially, there is a default population of codelets in the coderack. They are low-level *scouts*: In this work, their assignment is to build a description of the geometric figures in a Bongard problem by identifying specific *features* of the polygons, such as *shape*, *position*, and *size*. Each of these primitive concepts is a node in the concept network (I will use hereafter the terms "concept" and "node" interchangeably). The concept network is the system's body of prior knowledge. It includes all the concepts that are known to the program regardless of the specific problem it is dealing with. Concepts that are semantically related are connected by a link (the closer the relationship, the shorter the link), which allows the nodes to exchange *activation*.

Each node in the concept network has an activation value. A node is assigned some amount of activation when the program discovers in the problem instances of the concept it represents. An active node can spread activation to its neighboring nodes, depending on the length and nature of the connecting links. A node's activation decays at each time step, unless it is reinforced by codelets finding new or preexisting instances of the concept it represents. An active node therefore represents a concept that has a number of instances in the problem, and should be important in future actions of the system. In fact, active nodes can post codelets that seek other instances of the same concept.

Different nodes have different rates of decay, because some concepts are more complex than others and the system must be allowed to "study" them for a longer time, once they have become relevant. In the previous example, if the system discovers that a certain polygon is a triangle, both the nodes *triangle* and *shape* will become active and begin to urge the system to look for other instances of their concepts in the problem. But shape is a more general concept than triangle (in fact, we say that “triangle” is an *instance* of the *category* “shape”), and the program is generally biased towards pondering more general concepts for longer. Following Hofstadter’s terminology, *shape* has a greater *conceptual depth* than *triangle.* The program’s bias in favor of more general concepts is obtained by giving deeper concepts lower activation-decay rate. Defining the concept network therefore requires a set of concepts, a set of associated codelets, the structure of the links between them (including the link lengths), and the conceptual depth of each node.

A correct balance of top-down and bottom-up processes is crucial to the success of the program. Particularly, the program must initially carry out a wide-range *exploration* of the input, but it should later *exploit* the information it gains as its understanding of the problem progresses. A key role in this control of resource allocation is played by a variable called *temperature*. The temperature is described in detail in Section 7.5. For now, it will suffice to say that the temperature is computed at regular intervals by the program and it is defined in such a way as to represent a quantitative estimation of the quality and level of perceptual organization reached by the program. It is higher when the codelets have not been able to discover much *structure* in the problem, while it decreases when the codelets seem to have found a promising path toward the solution. By “structure” here I mean any element that the program has been able to add to the initial representation of the problem, such as the description of the shape or size of an object, or the realization that two different objects have the same shape. The temperature is used by the system to “tune” its degree of non-determinism, making it progressively more focused on the more promising structures discovered in the workspace (*i.e.*, making top-down pressures progressively prevail on bottom-up processes).

7. System implementation

7.1 Motivation, problem statement, and original contribution

The author’s interest in this project is mainly due to the appeal of the similarities between the behavior of complex adaptive systems and some of the principles of the theory of high-level perception (see Section 6.1) and the idea that the latter could be at the heart of the supreme performance of human beings in cognitive tasks. By carrying out this project, the author expected to achieve the following goals:

- To explore the challenges of designing and implementing such an architecture;
- To gain a sense of what the main factors are that drive the program towards a solution;
- To put such factors into a more general perspective, hoping to abstract the basic ideas that can be applied to the development of programs based on the same model, but applicable potentially to different domains.

Building a program capable of solving a Bongard problem with generic objects (such as those in Figure 3) using the bare binary bitmap images as input would be equivalent to solving a great part of the image-understanding problem, which obviously falls well beyond the scope of a Master’s thesis (Foundalis' program Phaeaco [19] — to my knowledge the most comprehensive Bongard-problem solver — could only solve a very limited subset of the original 100 problems). As stated in Section 6.2, the goal of this thesis was to design and implement a system that applies Copycat's model of analogical reasoning to a problem that is markedly different from the letter-string analogy puzzles Copycat deals with.

These considerations, together with compliance with design requirement 6 in the list of Section 6.2, led me to define a *set of "simplified" Bongard problems* which share a *general strategy* for their solution. The set of problems is defined by the following characteristics:

1. The problems contain only objects in the form of polygonal figures, namely triangles and quadrilaterals.
2. Each frame contains only one object.

The two characteristics enforce a very significant limitation in the complexity of the task the program faces, but note that this limitation is mostly related to the *low-level* processing of sensory input. Particularly, they allow us to completely skip the segmentation problem (an as-yet-unsolved problem in computer vision) and to deal with objects that can be described in relatively simple terms.

The program's strategy of solving each problem is defined by a certain number of sub-goals. To achieve the sub-goals, the program must show to perform the following actions.

1. The program can identify a limited set of *features* in each object.
2. The program can establish *correspondences* between two objects in the same set, based on the features identified in them.

3. The program can build a *hypothesis* that explains what all the objects in the same set have in common, based on the correspondences it has established.
4. When the program has found two (non-contradictory) hypotheses (one for each set) it can build an *answer* to the problem.

The words written in italics are the different *conceptual structures* that the program can build, and represent the sub-goals mentioned above (see Section 7.3 for a detailed description of these structures).

In conclusion, the set of problems the program can solve and the general strategy it must follow define my system as a high-level analogical-reasoning system tailored to solving Bongard problems, rather than a complete Bongard problem solver. They also define the criterion for judging successful performance of the system: I will consider the system successful if it shows it can consistently solve a set of test problems. This will provide a sanity check, showing that all the components in the model work harmoniously towards the solution. The goal of this study is to provide a sound implementation of the *analogy-making as perception* model. Judging success is made easier in our case because when a Bongard problem is correctly designed, there is only one possible answer, so either the program can discover it, or it cannot, in which case it fails.

For completeness, I note here that Copycat includes a *group* structure, which allows the program to do something like what needs to be done to see a triangle in the upper-left frame of *Group a* in Figure 1. This structure is not implemented in my system because (a) grouping in a Bongard problem is more complex than in Copycat's domain (see points 2 and 3 in the list below), and (b) groups represent a special case of perceptual structure in that the program may sometimes need to see them as composite *objects* (*e.g.*, the triangle mentioned above, which is obtained by grouping circles). Addition of a grouping mechanism should of course be a top priority in the list of future developments.

Finally, to better clarify the original contribution provided in this work, as well as some of its limitations, I briefly analyze here the differences between Copycat's domain and the domain of Bongard problems.

1. Copycat's problem always provides a *reference* to the solver in the form of the pair (*initial string, modified string*). The solver uses this hint to interpret the target string and explore its possible modifications. Bongard problems are more subtle in this respect, because no clear reference is provided: the solver must find motivation for rejecting or accepting a hypothetical solution by searching the problem for clues supporting each possibility.
2. Copycat's object-grouping mechanism is based on *adjacency*: A letter can be "bonded" only to letters that are adjacent to it in its string and if this happens, both letters may later become part of a group. Exploiting adjacency allows Copycat to limit the number of potential alternatives to explore for grouping. In Bongard problems grouping is not necessarily related to adjacency. For instance, in problem BP#58 (see 0) one needs to group two black squares of equal size in the left set and two black square of different size on the right. Grouping is in this case based on *shape*, *color* and *size*, with no way of reducing *a priori* the possible candidates for grouping on the basis of relative distances.
3. Copycat can explore relationships between adjacent objects within the same string and between objects in the initial and target string (analysis of the modified string is limited to a comparison

with the initial string). In a Bongard problem, relationships can be built at several levels among objects within the same frame (e.g., the circles in the upper left frame of P1 in Figure 1 can be grouped to form a triangle, and if the frame contained a second "normal" triangle, the two triangles could later be paired to form a shape-based group), objects in different frames in the same set, and even objects in different sets. There is a greater richness in the kind of relationships among objects that can be built in a Bongard problem.

4. The problems solved by Copycat and my system are *structurally* different. In Bongard problems there is no obvious counterpart of Copycat's strings and most of the perceptual structures the program builds.

These general differences, together with the obvious differences between a letter-string domain and a geometric-object domain, required rewriting and modifying many of Copycat's original components. Differences between my system and Copycat include all the codelets responsible for discovering and building perceptual structures in the problem, the definition of these structures, the set of concepts defining the program's prior knowledge and the machinery that manages objects and perceptual structures in a problem that is structurally different. Furthermore, for this project Copycat has been ported from CLISP to Java, with significant modifications, and a graphical interface has been developed in MATLAB®. The graphical interface allows one to monitor the behavior of the system during a run, showing plots of the temperature and codelet-urgency time histories, the activations of nodes in the concept network, the coderack inventory and tables of all the conceptual structures the program has built.

7.2 The concept network

Figure 4 shows the system's concept network, which represents the body of prior knowledge available to the program. As briefly explained in Section 6.3, the network is defined by a set of *nodes* (each with an associated *conceptual depth*), and a set of *links* between the nodes. Each node represents a concept and the links indicate semantic relationships between concept pairs. The current implementation has 28 nodes divided into five main categories as the following list shows.

- Category names:

  Geometric entity, feature-type, feature-value, label.

- Geometric entities:

  Polygon, vertex, edge, line-segment.

- Feature types and values (the values are listed in parenthesis after each type's name):

  Color (filled, outlined), horizontal position (left, horizontal center, right), vertical position (low, vertical center, high), shape (quadrilateral, triangle), size (small, medium, large).

- Labels:

  Same, opposite.

The *conceptual depth* is a value associated with each node. In general, concepts representing categories, being more general, have greater conceptual depth than their instances. The main role of this parameter is to influence the activation decay of its node. *Activation* is a variable associated with each node and represents the importance that the program is currently giving to the concept. Activation can have any integer value in the range 0 (no activation) to 100 (*full* activation), and the higher the value, the more relevant the concept is considered by the program. A node's activation changes during the run as a result of the scouting activity of agents whose function is to identify concept instances in the problem at hand. Concepts that have more instances receive greater activation. The program periodically updates the activation of each node in the concept network taking into account both the positive contribution received by the discovery of a node's instances in the problem and the negative contribution due to activation decay.

A node's activation decays with time at a rate that depends on its conceptual depth. More specifically, at each update a node loses a percentage of its activation equal to 100 minus its conceptual depth. Therefore, nodes with higher depth decay more slowly. Since I assign greater depth to more general concepts, these tend to remain active for longer than more specific concepts (*e.g.*, their instances). This means that when the program discovers a general concept (*e.g.*, *shape*), it will tend to "ponder" it for longer than its instances (in this case, *triangle* and *rectangle*). This behavior is in general desirable in Bongard problems, because they are all based on the contrast between two different instantiations of the same concept, so it is the latter that the system needs to focus on. When activation reaches a threshold value, currently fixed at 50, the node has a certain probability (which depends on the current temperature) to snap to full activation.

Nodes that are semantically related (e.g., a category and its instances) are connected by *links*, which act as paths through which a fully activated node can spread activation to other nodes. When fully activated, feature-type nodes can post *top-down* codelets to the coderack. These codelets are aimed specifically at seeking instances of the concept that posted them and, as explained in Section 7.4, constitute the main mechanism through which the program focuses its attention on some specific concepts, if they appear to be promising.

7.3 Perceptual structures

Problem BP#6 from Bongard's original collection (see Figure 5) has all the characteristics listed in Section 7.1 and therefore, besides including it in the set of test-problems for evaluating the performance of the program, I use it here as an example to illustrate the system. At the beginning of a run, the program receives as input the description of the problem, which includes one polygon per frame, with its edges and vertices. As stated in Section 7.1, a complete Bongard-problem solver would receive only the raw bitmaps, not structured input. This kind of input is acceptable in the case of a high-level analogical-reasoning system. These polygons are "atomic" to the system, which means that although the program has the notions of *line-segment*, *edge* and *vertex*, it will not attempt to break a polygon into its elementary components (in Copycat the "atoms" were the single letters making up each string). From now on, I will call these "atoms" *objects*. The program can build a number of *perceptual structures* (more briefly, *structures*) on top of the given objects: *features*, *correspondences*, *hypotheses* and *answers*. These represent what the program "understands" as it studies the problem, and form a hierarchy of constructs in that correspondences are based on features, hypotheses on correspondences and answers on hypotheses.

A *feature* is the most elementary structure the program can build, and it can be viewed as a (*Type*, *Value*) pair, where the type and the value are determined by the structure of the concept network *size* (see Section 7.2). The feature types available to the system are *color, horizontal* *position, vertical position, shape*, and *size* (see Section 7.2 for the complete list of the values). For the program, "identifying an object as a rectangle" means adding the pair (*shape, rectangle*) to its feature list. The type *color* is an exception in that it is attached by default to each object when the input is prepared, together with its appropriate value (*filled* or *outlined*). I justify this by noting that for these objects color is a rather basic feature that humans seem to immediately and effortlessly grasp. The existence of such "automatically activated" descriptors is described by Barsalou [27] in his theory of perceptual symbol systems as a characteristic of the process through which humans build mental representations. Of course, the presence of the *color* feature in the input does not represent a commitment for the program to consider this feature important throughout the entire study of the problem. The system can discover all the other features and use those that appear more meaningful in its path towards the solution.

Besides trying to describe each single object through its features, the program can also build *correspondences* between two objects within the same set. If two objects happen to share one or more features, the program will try to establish a correspondence between the two. As explained in Section 2, solving a Bongard problem requires identifying some key concept that is common to all the objects in each set. If two objects have one or more features in common, these may contain the concept that describes all the objects in the set. The program acknowledges this by building a correspondence.

I call the process through which the system identifies the features(s) that are common to all the objects in a set *hypothesis building*. A *hypothesis* is another perceptual structure that the program can build and represents an idea the program has developed about how to explain what all the objects in a set have in common. Borrowing from what seems to be the mechanism humans employ in studying a Bongard problem, the system can build a hypothesis starting from an object and analyzing its correspondences with other objects. If an object is involved in correspondences with one or more other objects, the correspondences are analyzed to determine whether among the features upon which they are based there are one or more that are common to all the correspondences. If at least one feature is common to all the correspondences, it is used as the basis for the proposal of a hypothesis to explain the entire set. For reasons that I will explain in detail in Section 7.4, the program will *not*, in general, proceed immediately to verify this hypothesis. But when this happens, a check will be performed on all the objects in the set to see if the features upon which the proposed hypothesis is based have been discovered in *all* the objects in the set. Only if this check returns a positive outcome will the hypothesis be accepted as satisfyingly describing the set.

The last kind of perceptual structure the program can build is the *answer*. The system can build an answer only if it has previously built the two hypotheses for each of the two sets. After the second hypothesis has been discovered, one of the decentralized self-watching mechanisms in the system urges it to consider soon the possibility of building an answer. When this happens, the program checks the (sets of) types and values the two hypotheses are based upon, identifies the feature type that has different feature values in the two hypotheses and uses these values to build the answer. This process is relatively straightforward due to the nature of Bongard problems, for which there must only be one possible answer. In other words, when a problem is correctly designed, there is no ambiguity about the solution: If one is devised, it is no doubt the correct one.

7.4 Codelets, coderack and more on structure building

The *coderack* and the *codelets* have been briefly introduced in Section 6.3. The program has a main loop that runs a single codelet per iteration (hereafter also called *time step*). The coderack is a structure hosting the codelets to be run, grouped by urgency. There are four different categories of codelets, each with different functions: *scout*, *strength-tester*, *builder* and *breaker*. Scout codelets can be *bottom-up* or *top-down*. Bottom-up scouts are aimed at seeking a certain kind of structure in the workspace (e.g., they will try to identify a feature, or to find a correspondence), but their search is not influenced by the state of activation of nodes in the concept network. On the other hand, top-down scouts are posted to the coderack on account of the existence of fully-activated nodes and represent a mechanism through which the system focuses its attention on concepts that appear particularly relevant and whose instances should therefore be sought with greater determination.

It is easier to explain the reasons for these different categories through an example of how they are employed during a run of the system. This requires a short introduction to the *temperature* (this important variable is described in detail in Section 7.5). For the time being, it will suffice to say that the temperature is periodically updated by the program during a run, and provides the system with a rough measurement of the quantity and quality of the structures currently present in the problem. A high temperature indicates that the program is not very satisfied with the structures it has built. The temperature decreases when the program has reasons to believe it is moving in a promising direction (the reason for this inverse relationship between temperature and the program's judgment of its performance will become clear in Section 7.5).

As an example, let us consider problem BP#6 (see Figure 5), whose solution is “triangle vs. quadrilateral”. In the beginning, the coderack is filled with scout codelets. Most of these are targeted at seeking *features* (see Section 7.3), and a few at seeking correspondences. This initial bias towards seeking features is meant to help the program identify and build the most elementary structures, upon which all the others are based. Identifying a potential feature is equivalent to finding in the problem instances of concepts that are represented in the concept network by *feature-value* nodes (see Section 7.2). This is only the first step towards *building* the feature, which requires the successful completion of a sequence of three operations, each carried out by a different kind of codelet: the *scout*, the *strength-tester* and the *builder* (this scheme might appear excessively complex, especially in the case of rather elementary problems like this one, but the rest of this section will clarify that it is a crucial part of implementing high-level perception).

At the beginning of a run the temperature is at its maximum value (indicating that the program has not yet built any significant structures in the problem), and all the codelets in the coderack are equally probable to be chosen for running. At each step, a codelet is chosen probabilistically from the coderack and run. All the initial feature scouts are bottom-up, which means that they are not instructed to seek features of a specific type. When a scout runs, it picks an object in the workspace and a feature-type (e.g., *shape*) and tries to see whether it is possible to describe the object in terms of this feature type. At this moment, all objects are equally likely to be chosen, but as the run progresses the program tries to make more targeted choices, and objects that are involved in strong correspondences or that have relatively few features attached become more likely to be picked. A certain amount of initial random exploration is consistent with experimental evidence that human viewers begin to examine a novel scene by randomly (and quickly) moving the focus of attention to many different locations (see, for instance, [8]).

The feature scout identifies the feature value that best describes the object (for instance, *triangle*, if the object belongs to the left set of BP#6) and posts a *proposed* feature to the workspace and a *strength-tester* codelet to the coderack. For a structure, being *proposed* means being considered by the program as *potentially* important, but only structures that achieve the status of *built* are considered by the program as actually relevant to the problem. The urgency of this *follow-up* codelet (the strength-tester) is an increasing function of the activation of the feature-type node (*i.e.*, the codelet has a higher probability to be run soon if the feature type currently appears to be relevant in the problem). If chosen for running in the future, this codelet will check whether the *context* supports the idea that the given object should be described as a triangle. This evaluation is summarized in a measure called the *strength* of the feature, which has both an *internal* and an *external* component. The *external* component is computed in the same way for all features. It is proportional to the activation of the feature value and also increases with the number of other features in the workspace with the same type and value. The *internal* strength is computed in different ways depending on the particular feature, but in general it is an estimate of how close the object is to an "ideal" object that would be "perfectly" described by the feature. The codelet then makes a probabilistic decision as to whether the program should keep the proposed feature (*shape, triangle*), possibly turning it into a built structure in the workspace at some future step. The decision is probabilistic: the higher the strength, the higher the probability that the feature will pass the test. A successful outcome results in the sending of activation to the *triangle* node and the posting of a feature builder on the coderack, with urgency an increasing function of the strength computed before.

Node activation and the posting of follow-up codelets are extremely important for the self-regulation of the system. The more active a node is, the more it will try to influence the activity of the program. While the activity of bottom-up scouts is driven only by local considerations on the objects in the workspace, an active node in the concept network can exert a top-down *pressure* on the system by posting codelets that look for specific instances of the concept itself. The more active the node, the higher the urgency associated with these codelets.

As the program progresses in its analysis of the input, nodes corresponding to the concepts identified by codelets become activated and begin to influence the behavior of the program by posting targeted codelets with higher urgencies. As more instances of a concept are found, the program is forced to progressively focus its attention on that concept. The search, from completely random, becomes progressively more deterministic. Existing instances of the concept also cooperate to support the building of new instances by adding to the strength of any proposed new instance (contextual pressure). Conversely, if the activation of a concept is not continually sustained by other found instances, it progressively decays, weakening the relevance of the concept in future actions by the program.

At each step, the choice of which codelet to run is probabilistic. Low-urgency codelets (corresponding to low-activation, less frequently instantiated concepts) have a lower probability of running than high-urgency codelets, but this probability never drops to zero. In general, it is unlikely that the sequence scout → tester → builder for a certain structure will be completed in three consecutive time steps. The execution of the three codelets will be interleaved with that of other codelets working on different structures. But a sequence of high-urgency codelets will on average be completed more quickly than a sequence whose codelets have lower urgency. At every time step, many different structures exist, at different stages in their development. Progress can always be made on the low-urgency paths: it just is slower.

In our example, the third and last stage of the sequence of codelets is the running of a *builder* codelet. In the case of an elementary structure like a feature, this codelet will basically proceed to build the feature. But for more complex features the process is less straightforward. Consider, for instance, a run in which a correspondence based on the feature (*size, small*) has been built between the two smaller black triangles in the left set of BP#6. This could have happened because, for instance, these two triangles may have been picked and described as small at some early step in the run, when there was still no reason to believe *size* was less important than any other feature type. After some time, other concepts (*shape* and *triangle*, for instance) may have emerged as relevant (to the program, a concept is *relevant* if its node is fully activated), whereas *size* may have lost its full-activation state. If this is the situation when the builder tries to build the *size*-based correspondence, the attempt will fail and the structure will not be built. Since a structure becomes visible to other codelets only when it is *built* (as opposed to be just *proposed*), the correspondence will not have any weight in any future decisions the program makes (*e.g.*, it will not be able to support the proposal of other correspondences based on the *size* concept). Before a correspondence can be built, the codelet will also check whether the objects involved are already part of other correspondences that are incompatible with the one it is trying to build. Two correspondences are compatible only if they are based on different feature types, or, if they are based on the same type, when they also have the same feature value (*i.e.*, the program can currently establish only *sameness* correspondences). If any incompatible correspondences exist, the proposed correspondence will have to *fight* them and only if it wins all the fights will it finally achieve the *built* state. The mechanism of a fight is the same for all structures: A probabilistic tournament in which the structure with highest strength has a greater probability of survival (differences in strength are amplified when the temperature is low).

This example helps illustrate two reasons for breaking up the process of building a structure into three steps. First, it implements the principle of *parallel* processing by elementary agents that was introduced in Section 6.1: At any given step in the run, many different structures at different points in their development coexist in the workspace. Secondly, this mechanism allows the system to dynamically and efficiently allocate resources (*i.e.*, codelets) to different paths along the way to the final solution, through the balancing of *top-down* and *bottom-up* pressures, as required by the theory of high-level perception (see Section 6.1). The importance of this is apparent in the example illustrated above: the size-based correspondence, proposed initially by a bottom-up scout, will not be pursued as a result of the top-down pressure from the concept network urging the system to give precedence to the evaluation of other concepts. This is generally useful because solving a Bongard problem often requires information that is not immediately apparent. Problem P2 in Figure 1 is another good example. Let us assume that one begins to parse the problem starting from the upper-left-corner frame. They immediately recognize several circles, which seem to constitute the most relevant source of information in the frame. If we do not examine this frame in the *context* of the entire left set, the fact that the circles' centers appear to be aligned on the edges of an imaginary triangle may not be deemed relevant at first. But when one detects the presence of triangles in all the other frames on the left side, one might immediately remember having "seen" a triangle in the upper-left-corner frame: One just postponed the evaluation of the relevance of this idea, because at first it seemed less *urgent* than others. This example shows the essential role of analogy-making in solving Bongard problems: We can solve it because we can map the group of circles to an abstract notion of "triangle".

As the system advances in building higher-level structures out of lower-level ones, the temperature decreases, reflecting the increasing amount of perceptual organization achieved. As a result, in every probabilistic decision, the outcome with higher probability receives some "boost" at the expenses of lower-probability outcomes, and the differences in codelet urgencies are amplified. The system tends to explore promising paths more consistently, but can still, with lower probability, consider alternative solutions. Important aspects that so far have been neglected can still be discovered and exploited, if context urges the system to do so.

7.5 Temperature

Section 6.3 mentioned that balancing top-down and bottom-up processes is crucial to the success of the program. The program needs a way to assess the current state of its search for a solution. It must be able to *shift* from searching for generic information (prevalence of bottom-up codelets in the early stages, when little is known about the problem) to focusing primarily on exploiting information that is likely relevant to the particular problem. This is achieved by means of a global variable called *temperature*, which is defined in such a way as to represent a quantitative estimation of the quality and level of perceptual organization reached by the program. The temperature is computed at regular intervals during a run using a rather lengthy formula ultimately based on the strength of the hypotheses (if any exist) and the number of fully-active feature values in the built features attached to every object. It is higher when the codelets have not been able to discover much structure in the problem, and it decreases when the codelets seem to have found a promising path toward the solution. The temperature provides a feedback mechanism between the programs's understanding of the problem (*i.e.*, the structure the system has built on top of the original input) and the degree of non-determinism of its actions, making the system progressively more focused on developing the more promising structures discovered in the workspace. Temperature affects the following decisions:

- Probability of posting scout codelets for a given structure type.
- Probability for a posted codelet of being replaced in the coderack by a new one.
- Urgency of bottom-up codelets.
- Whether to break an existing structure.
- Strength computation for structures.
- The object chosen as target by a scout codelet.
- The result of fights between structures.

Temperature has therefore a crucial role in the program's strategy for allocating resources in its search for a coherent set of structures to solve the problem. A high temperature makes the system more inclined towards a wide-range *exploration* of the input whereas low temperature causes the program to *exploit* more deterministically the information that has been made available through its understanding of the problem (*i.e.*, the clues given by the structures it has built). The idea is that this strategy of using information as it becomes available to bias probabilistic decisions (therefore never ruling out completely any possible path of exploration) is the key to guide a system towards success when time and resources are limited and the number of paths to explore is intractably large. Although the set of problems this system can deal with probably is not challenging enough to fully verify this claim, temperature is a key feature of analogy-making as perception that this implementation definitely could not leave out.

8. System's performance on the test problems

8.1 Introduction

This section illustrates the performance of the system on a set of 6 test problems. They have solutions based on 4 of the 5 feature types available to the system (type *vertical position* is not present because the algorithm that identifies this feature is exactly the same as the one for *horizontal position*). Two kinds of tests were performed: Baseline and lesion. In the *baseline* tests, the program was run 1000 times on each of the 6 test problems. In the *lesion* tests, the program was modified to change some relevant aspect of the cognitive model and run again 1000 times on problems BP3, BP6, BPH and BPS (BP3.1 and BP6.1 were skipped because they represent just variations on BP3 and BP6). The lesions were:

- Temperature clamped at 0 (minimum value).
- Temperature clamped at 50.
- Temperature clamped at 100 (maximum value).
- Suppression of the differences in conceptual depth (all nodes in the concept network were given a conceptual depth of 50).

The program reached the correct solution in all of the runs, which meets one of the criteria for judging success identified in Section 7.1. The number of codelets run to reach the solution was chosen as the figure of merit to evaluate the performance of the program. Sections 8.2 through 8.7 show the test problems, whereas Sections 8.8 and 8.9 analyze the results of all the experiments.

8.2 BP3 — *Filled vs. outlined*

(Figure 7 HERE)

This is a new rendition of BP#3 from Bongard's original collection [9].

8.3 BP3.1 — *Filled vs. outlined*

(Figure 8 HERE)

This is the same problem as BP3, but the objects on the right are also positioned in the middle of the horizontal axis. This modification makes it possible to build two different hypotheses to explain what the objects in the right set have in common, one based on (*color*, *outlined*) and one based on (*position*, *horizontal center*). If the program built this alternative hypothesis, its erroneous nature could only be discovered if the program also tried to build the correct (*color*, *outlined*) hypothesis at a later stage, or if it later built the only possible hypothesis for the left set, based on (*color*, *filled*). In the first case, the new hypothesis for the right set would have to fight the preexisting one. In the second case, a high-urgency answer-builder codelet would be posted. When run, the codelet would notice the incompatible nature of the hypotheses for the two sets (they are based on different feature types) and, after a fight, one of the two would be destroyed. The modification was introduced to see whether it would cause some noticeable difference in the program's behavior.

8.4 BP6 — *Triangle vs. quadrilateral*

(Figure 9 HERE)

This is a new rendition of BP#6 from Bongard's original collection [9].

8.5 BP6.1 — *Triangle vs. quadrilateral*

(Figure 10 HERE)

This is the same problem as BP6, but the objects on the right are also positioned in the middle of the horizontal axis. As explained in Section 8.3, the modification has potential for making the problem harder and is introduced to see whether it causes some noticeable difference in the program's behavior.

8.6 BPH — *Left vs. center*

(Figure 11 HERE)

This problem is inspired by BP#8 from Bongard's original collection [9], which is also based on horizontal positioning.

8.7 BPS — *Small vs. large*

(Figure 12 HERE)

This problem is inspired by (and has the same solution as) BP#2 from Bongard's original collection [9].

8.8 Baseline tests

The main purpose of these experiments was to show that the system can consistently solve problems based on each of the different feature types in the concept network. Besides the correctness of the answer, in a system like this, with neither a centralized controller nor a deterministic algorithm, we can check for the correctness of the behavior of the program only indirectly. As indicators of the program's behavior, I monitored in particular the time histories of the temperature and of the codelet urgencies during a run. Finally, these experiments provided some insight into aspects of the architecture and the particular way they were implemented in this system, suggesting directions for improvement and future developments.

The *temperature* represents the program's assessment of the quality and quantity of the structures it has built, and we expect this variable to be consistent with the observed structure-building/breaking events. In general, the temperature will start at its maximum value, because at the beginning of a run there is no structure built by the program. The temperature should then show a generally decreasing trend as the program builds structure into the problem, with more pronounced negative (positive) slope when high-level structures are built (broken).

The *codelet urgencies* are relevant because, when examined separately for each kind of structure the codelets work on, they show what the program is currently focusing its attention on at a particular moment during the run. Since there is no central controller determining what the program should do at each time step, the allocation of resources (*i.e.*, codelets) is the emerging result of the assessment by the program of the structures it has built (*i.e.*, their *strength*) and of the concepts that appear to be most relevant at any particular moment (*i.e.*, the *activations* of the nodes in the concept network). If the implementation of these decentralized control systems is sound, we expect the program to start focusing on the lowest-level structures (*i.e.*, features) and progressively shift towards higher-level structures. Hypotheses scouts should gain relevance (*i.e.*, urgency) only after a while, when some correspondences have been built, and answer builders will be the last codelets to make their appearance at the highest urgency level.

The graphical interface proved useful to check that the temperature and codelet urgencies actually showed the desirable properties described above. Figure 13 shows the time histories of the codelet urgencies (grouped by structure category) and the temperature and node activations at the end of one of the baseline runs on problem BP6 (“triangle vs. quadrilateral”); the trends visible in these plots are quite typical of the baseline experiments.

The temperature plot shows the desired behavior: The temperature starts high and then progressively decreases as the program progresses into the run. By observing the lists of structures built by the program (not shown in the figure) one can see that the temperature decrease is mainly driven by the appearance of built correspondences. The temperature at the end of the run is very low (6), showing that the system was very satisfied with the structure it had built.

The urgency plot reports the fraction of the total urgency in the coderack held by each type of scout codelet (feature, correspondence, and hypothesis), answer-builder codelets and breaker codelets. It also reports the number of codelets in the coderack. The current value of the total urgency in the coderack is very high because the plot refers to the end of the run, when the temperature is very low. In these conditions, the differences among the urgency values the program computed at the time each codelet was posted are strongly amplified, particularly for the most urgent codelets. This makes it much more likely for the system to choose for running codelets that, when posted, where deemed to be particularly promising (and therefore got higher urgency values). This is the mechanism through which temperature influences the focus of attention of the program.

All the urgency plots in Figure 13 show some oscillations, which are due to the probabilistic nature of the program’s algorithm, but also allow abstracting some general trends. The program shows a generic interest in looking for features throughout the entire run, with some decrease towards the end. Correspondences are initially pursued with an interest similar to that of features, but then decline more quickly. The program's interest in building hypotheses is generally dependent on temperature: As the temperature decreases, the program shows greater urgency for hypothesis scouts. This is correct, because the decrease in temperature indicates that a good amount of correspondences are present, which are used as basis for the hypotheses. Finally, interest in building an answer only appears very close to the end, and immediately very high: That is the moment when the system had two built hypotheses in place (one for each set) and was therefore ready to build the answer.

Overall, these plots show the desired behavior for both the temperature and the codelet urgencies. As an additional confirmation of the general sanity of the system, the node activations in Figure 13 show that the nodes *shape*, *triangle*, and *quadrilateral* are fully active, which is consistent with the correct answer for this problem (fully activation of other nodes such as *same* or the categories *feature type* and *geometric entity*, together with the latter's instantiations, is also something we expect in the problems of this test set).

Figure 14 shows boxplots of distributions of the number of codelets run for each of the 6 test problems. The rectangular boxes cover the interquartile range (*i.e.*, the edges of the box are the 25^{th} and 75^{th} percentiles). The vertical line inside the box indicates the mean value. The "whiskers" extend from the box edges to the farthest data point within 1.5 times the interquartile range. Red-cross markers indicate outliers, defined as data points that fall outside the range of the whiskers.

Outliers, some of which rather severe, are present for all tests beyond the upper quartile mark. Since all the problems essentially test the ability of the program to identify the feature type on which the final answer must be based, a longer time to reach the solution in general indicates some trouble in building the features of that type. This can be due both to the probabilistic nature of the program's choices (in some unlucky cases the program just does not happen to ever consider something important) and to the need for further tuning of some of the algorithms (see the discussion below about the algorithm for assessing an object's size). But, in general, the system would benefit from a mechanism allowing it to realize when it has been wandering without making much progress for a long time. When this happens, the program should stop focusing on the structures it has built (which apparently have not been able to promote significant progress) and start again a "broad'spectrum" search by running a number of bottom-up agents. This feature was implemented in Copycat and these results provide evidence of its importance in the implementation of the perceptual approach to analogy-making.

Table 1 lists the mean and standard deviation of the figure of merit for all the baseline experiments. Looking at the mean values, a certain variability in the program's performance on different problems is apparent. For problems BP3 and BP3.1 ("filled vs. outlined", see Figure 7 and Figure 8) the program reached the solution much faster than for any other case. This is due to the nature of the input the program receives, which includes the *color* feature (see Section 7.3). This means that the program can potentially start building correspondences based on *color* from the first time step, therefore speeding up the entire process. BP3.1 is in theory more complex (see Section 8.3), but the results show no difference in the two mean values at any reasonable significance level ($p = 0.42$): In this case, the building of the correct, color-based correspondences usually prevails very quickly.

As for the results on the other test problems, assuming (quite arbitrarily) BP6 as reference, the null hypothesis of the mean values of the other test experiments belonging to the same population was tested by means of the standard two-sample *t* test. BPH is the only one whose mean turned out to be not significantly different from that of BP6 at any reasonable significance level. The conclusion is that the performance of the program *does* show some dependence on the particular problem it works on.

The results for BPS ("small vs. large", see Section 8.7) show a mean value about 16% higher than BP6 and BPH. This draws attention to a detail that probably requires some tuning in the program. As explained above, for these kinds of problems a longer time to reach the solution in general indicates some trouble in building the features of that type. Furthermore, Section 7.3 showed that for any structure the probability of being built is a growing function of its strength. For quantitative features, the strength is computed on the basis of a *distinguishing parameter*, which in the case of the feature type *size* is the *area ratio* — *i.e.*, the ratio of the area of the object to the total area of the frame. Two thresholds are set to divide the range of possible values into three subintervals, to which the feature values *small*, *medium* and *large* are assigned. The strength of the feature is highest when the area ratio falls in the middle of the subinterval, and decreases as the area ratio gets closer to the extremes of its subinterval. The slope of this decreasing function is probably too high, so size-based features too often turn out to be too weak to be built, delaying the discovery of the right solution in this particular case.

8.9 Lesion tests

8.9.1 Introduction

The purpose of the lesion experiments is to observe the role played by some components of the model in the program's functioning. This is done by removing or modifying a specific component of the model and observing the influence this has on the program's behavior. The components chosen in this work are *temperature* and *differences in node conceptual depths*.

8.9.2 Temperature lesion

The temperature is an indication of the program's assessment of the quality and quantity of structure it has built in the problem, and it is described in detail in Section 7.5. This variable is used in all the probabilistic decisions made by the program. In general lower temperature values make the system more inclined towards selecting outcomes that appear more promising. Fixing the temperature at a high value therefore should make the program's decisions more random, whereas a low temperature value should induce the program to make very conservative decisions, almost invariably choosing what looks most promising. The temperature lesions consisted of clamping the temperature value at 0 (minimum value), 50 and 100 (maximum value). For each lesion, 1000 runs were performed on BP3, BP6, BPH and BPS.

Figures 15 through 17 show the results of the temperature-lesion experiments, whereas Table 2 reports the mean and standard deviation of the number of codelets run for each problem, together with the same parameters obtained in the baseline experiments (for comparison). The general trend visible in the distributions shows that clamping the temperature causes an increase in the mean number of codelets run, and the effect is more pronounced as the temperature value increases. With the notable exception of BP3, which shows a peculiarly bad performance at the minimum temperature value, the figure of merit is always highest in the case of maximum temperature. In general (with some mild exceptions), clamping the temperature also seems to cause an increase in the variance, this effect being more or less marked depending on the problem. Both these phenomena provide some evidence of the fact that a variable temperature helps the program build a set of structures that lead it to the solution somewhat more efficiently. Yet, the results appear less conclusive than those of a similar experiment performed with Copycat [30]. This may be an indication that although temperature is generally "well behaved" (as shown by the time-history plots), the formula for its calculation might not be ideal. On the other hand, it could also be that the set of test problems simply is not challenging enough, and the benefit of temperature-based resource-allocation for such easy problems is not dramatically apparent. This should be verified after the code is expanded to solve more complex problems, with higher numbers of objects and more complex structures, to avoid overfitting of the system to the limited test set currently available.

8.9.3 Conceptual-depth lesion

The conceptual depth of a node is an indication of how general the concept represented by the node is (see Section 7.2). It affects a node's activation-decay rate, the urgency of top-down codelets posted by a node and the strength of the (*feature-type*, *feature-value*) pairs upon which correspondences are built. In this case, the lesion consists in assigning the same conceptual depth to all the nodes in the concept network. Suppressing the differences in conceptual depth among the nodes should affect the building of top-down pressure by the program, *i.e.*, the influence of node activation on the system's actions. In this experiment, 1000 runs were performed on BP3, BP6, BPH, and BPS with all conceptual-depth values set to 50.

Figure 18 shows the distributions of the number of codelets run and Table 3 reports the mean and standard deviation of this figure, together with the values of the same parameters obtained in the baseline experiments (for comparison). In general, the mean values increase with the conceptual-depth lesion, whereas the standard deviations do not always follow this trend. The trend for the mean values supports the idea that conceptual depth differences help the program direct its efforts more efficiently towards the correct answer. As discussed in Section 8.9.2, more conclusive results would probably require a more challenging set of test problems.

9. Conclusions

The goal of this thesis was to build a system adapting the basic features of the *analogy-making as perception* model (Mitchell and Hofstadter [30]) originally implemented in Copycat to the markedly different domain of Bongard problems. The adaptation of this architecture to visual pattern-recognition problems has its theoretical foundations in the theories of interactive vision [8], high-level perception [11], and in general principles governing complex adaptive systems [33].

The parts of interest in Copycat's code have been ported from CLISP to Java, with significant modifications including: A completely new set of codelets for discovering and building perceptual structures in the problem; the definition of these structures; the set of concepts defining the program's prior knowledge; the machinery that manages objects and perceptual structures in a problem that is structurally different from Copycat's letter-string puzzles. Furthermore, a graphical interface has been developed in MATLAB® to monitor the program's functioning in real time for diagnostic purposes during testing and debugging. The graphical interface includes plots of the temperature and codelet-urgency time histories, the activations of nodes in the concept network, the coderack inventory and tables of all the conceptual structures the program has built.

The performance of the problem on a test set of 6 problems has been analyzed, showing evidence of a sound implementation of the model. The tests also provided indications of what the first additions to the current implementations should be (see Section 10). Overall, the system provides a concrete basis for a Bongard-problem solver based on the perceptual approach to analogy-making.

10. Directions of future development

As explained extensively by Linhares [18], the implementation of a complete Bongard-problem solver remains a task well beyond the capabilities of current artificial intelligence. Despite the potential of the perceptual approach to analogy-making, how to integrate low-level image processing into a perceptual model of analogical reasoning is still an open question, only partially solved even in Phaeaco [19]. It was not the goal of this thesis to make a significant step in this direction, but certainly this system leaves much room for improvement in terms of increasing the size of the domain of Bongard problems it can treat. Extending the system to include a general mechanism for grouping objects within a frame (as explained in Section 7.1) would definitely represent a very significant contribution in this sense.

In terms of completing the implementation of *analogy-making as perception*, a useful addition would be a mechanism allowing the system to realize when it has been working without making much progress for a long time. In this case, the program should break some of the structures it has built (which apparently have not been able to promote significant progress) and start again a “broad-spectrum” search by running a number of bottom-up agents. This feature was present in Copycat and the tests provided clear evidence of its importance in the implementation of the perceptual approach to analogy-making.

11. Acknowledgements

This work was partially funded by a grant from the J.S. McDonnell Foundation. I am grateful to my adviser Melanie Mitchell for her great patience and support.

12. References

1. Marr, D.: The Philosophy and the Approach. Ch.1 in Vision: A Computational Investigation into the Human Representation and Processing of Visual Information. New York, NY (USA): Freeman, 1982.
2. Canny, J.: A Computational Approach to Edge Detection. IEEE Transactions on Pattern Analysis and Machine Intelligence 8(6): 679–698, 1986.
3. Russel, R.J., Norvig, P.: Perception. Ch.24 in Artificial Intelligence, a Modern Approach. Upper Saddle River, NJ (USA): Prentice Hall, 2002.
4. Jan, T., Piccardi, M., Hintz, T.: Automated Human Behavior Classification Using Modified Probabilistic Neural Network. In Proc. Int. Conf. Computational Intelligence for Modelling, Control and Automation. Vienna, Austria, 2003
5. Plamondon, R., Srihari, S.N.: On-line and Off-line Handwriting Recognition: A Comprehensive Survey. IEEE Transactions on Pattern Analysis and Machine Intelligence 22(1): 63–84, 2000.
6. Tadeusiewicz, R.T., Ogiela, M.R.: Artificial Intelligence Techniques in Retrieval of Visual Data Semantic Information. In Menasalvas E. (editor): Atlantic Web Intelligence Conference 2003, pp.18–27. Heidelberg, Germany: Springer-Verlag, 2003.
7. Ogiela, M.R., Tadeusiewicz, R.T.: Nonlinear Processing and Semantic Content Analysis in Medical Imaging — A Cognitive Approach. IEEE Transactions on instrumentation and measurement 54(6): 2149–2155, 2005.
8. Churchland, P.S., Ramachandran, V.S., Sejnowski T.J.: A Critique of Pure Vision. Ch.2 in Koch C., Davis J.L. (eds): Large-Scale Neuronal Theories of the Brain. Cambridge, MA (USA): The MIT Press,

1994.

9. Bongard, M.M.: Pattern Recognition. Rochelle Park, NJ (USA): Spartan Books, 1970.
10. Foundalis, H.: http://www.cs.indiana.edu/~hfoundal/res/bps/bpidx.htm, 2006.
11. Hofstadter, D.R.: Fluid Concepts and Creative Analogies: Computer Models of the Fundamental Mechanisms of Thought. New York, NY (USA): Basic Books, 1995.
12. Hofstadter, D.R.: On Seeing A's and Seeing As. Stanford Humanities Review 4(2): 109–121, 1995
13. Linhares, A.: Book review: Fluid Concepts and Creative Analogies: Computer Models of the Fundamental Mechanisms of Thought. AI Communications 12:121–123, 1999.
14. Chalmers, D.J., French, R.M., Hofstadter, D.R.: High-Level Perception, Representation and Analogy: A Critique of Artificial Intelligence Methodology. Journal of Experimental and Theoretical Artificial Intelligence 4: 185–211, 1992.
15. Holyoak, K.J., Gentner, D., Kokinov, B.N.: Introduction: The Place of Analogy in Cognition. Ch.1 in Holyoak, K.J., Gentner, D., Kokinov, B.N. (eds): The Analogical Mind: Perspectives from Cognitive Science. Cambridge, MA (USA): The MIT Press, 2001.
16. Morrison, C.T., Dietrich, E.: Structure-Mapping Vs. High-Level Perception: The Mistaken Fight Over the Explanation of Analogy. In Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society, pp.678–682. Pittsburgh, PA (USA): Lawrence Erlbaum Associates, 1995.
17. Hofstadter, D.R.: Artificial Intelligence: Prospects. Ch.19 in Gödel, Escher, Bach: An Eternal Golden Braid. New York, NY (USA): Basic Books, 1979.
18. Linhares, A.: A Glimpse at the Metaphysics of Bongard Problems. Artificial Intelligence 121: 251–270, 2000.
19. Foundalis, H.E.: A Cognitive Architecture Inspired by Bongard's Problems. PhD thesis, Indiana University, Bloomington, IN (USA), 2006.

20. Kokinov, B.N., Petrov, A.A.: Integrating Memory and Reasoning in Analogy-Making: The Analogy-Making AMBR Model. Ch.3 in Holyoak K.J., Gentner D., Kokinov B.N. (eds): The Analogical Mind: Perspectives from Cognitive Science. Cambridge, MA (USA): The MIT Press, 2001.

21. French, R.M.: The computational modeling of analogy-making. TRENDS in Cognitive Sciences 6(5), 200–205, 2002.

22. Newell, A.: Physical Symbol Systems. Cognitive Science 4, 135-183, 1980.

23. Hofstadter, D.R.: Waking Up from the Boolean Dream, Or Subcognition As Computation Ch.26 in Metamagical Themas. New York, NY (USA): Basic Books, 1985.

24. Harnad, S.: The symbol-grounding problem, Physica D 42, 335–346, 1990.

25. Brooks, R.A.: Intelligence Without Representation. Artificial Intelligence 47, 139–159, 1991.

26. Forbus, K.D., Gentner, D., Markman, A.B., Ferguson R.W.: 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(2): 231–257, 1998.

27. Barsalou, L.W.: Perceptual Symbol Systems. Behavioral and Brain Sciences 22: 577-660, 1999.

28. Gentner, D.: Structure-Mapping: A Theoretical Framework for Analogy. Cognitive Science 7(2): 155–170, 1983.

29. Falkenhainer, B., Forbus, K.D., Gentner, D.: The Structure-Mapping Engine: Algorithm and Examples. Artificial Intelligence 41:1–63, 1989.

30. Mitchell, M.: Analogy-Making As Perception: A Computer Model. Cambridge, MA (USA): MIT Press, 1993.

31. Mitchell, M.: Analogy-Making As a Complex Adaptive System. In Segel, L., Cohen, I. (eds.): Design Principles for the Immune System and Other Distributed Autonomous Systems, pp.335–360. New York, NY (USA): Oxford University Press, 2001.

32. Erman, L.D., Hayes-Roth, F., Lesser, V.R., Reddy, D.R.: The HEARSAY-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty. Computing Surveys 12: 213-253, 1980.
33. Mitchell, M.: Complex systems: Network Thinking. Artificial Intelligence 170(18): 1194-1212, 2006.
34. Marshall, J.: Metacat: A Self-Watching Cognitive Architecture for Analogy-Making and High-Level Perception. PhD thesis, Indiana University, Bloomington, IN (USA), 1999.
35. Dor, U.: The ear's mind: A computer model of the fundamental mechanisms of the perception of sound. Master's thesis, Delft University of Technology, Delft, The Netherlands, 2005.
36. Bogner, M., Ramamurthy, U., Franklin, S.: "Consciousness" and conceptual learning in a socially-situated agent. http://www.msci.memphis.edu/~cmattie/Consciousness_and_Conceptual_Learning_in_a_Socially_Situated_Agent/Consciousness_and_Conceptual_Learning_In_A_Socially_Situated_Agent.html, 2009.
37. Saito, K., Ryohei, N.: A Concept Learning Algorithm with Adaptive Search. In Furukawa, K., Michie, D., Muggleton, S. (eds.): Machine Intelligence 14 — Applied Machine Intelligence, pp. 347–363. Oxford, UK: Clarendon Press, 1995.
38. Draper, B.A., Brolio, J., Collins, R.T., Hanson, A.R., Riseman, E.M.: Image Interpretation by Distributed Cooperative Processes. In Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR '88), pp. 129–135, 1988.
39. Slaney, M.: A Critique of Pure Audition. In Rosenthal D., Okuno H. (eds.): Proceedings of the Computational Auditory Scene Analysis Workshop, 1995 International Joint Conference on Artificial Intelligence, pp.13-18. Montreal, Canada , 1995.
40. Mitchell, M.: Self-Awareness and Control in Decentralized Systems. In Working Papers of the AAAI 2005 Spring Symposium on Metacognition in Computation. Menlo Park, CA: AAAI Press, 2005.
41. French, R.M.: The subtlety of sameness: A theory and computer model of analogy-making. Cambridge,

MA (USA): The MIT Press, 1995.

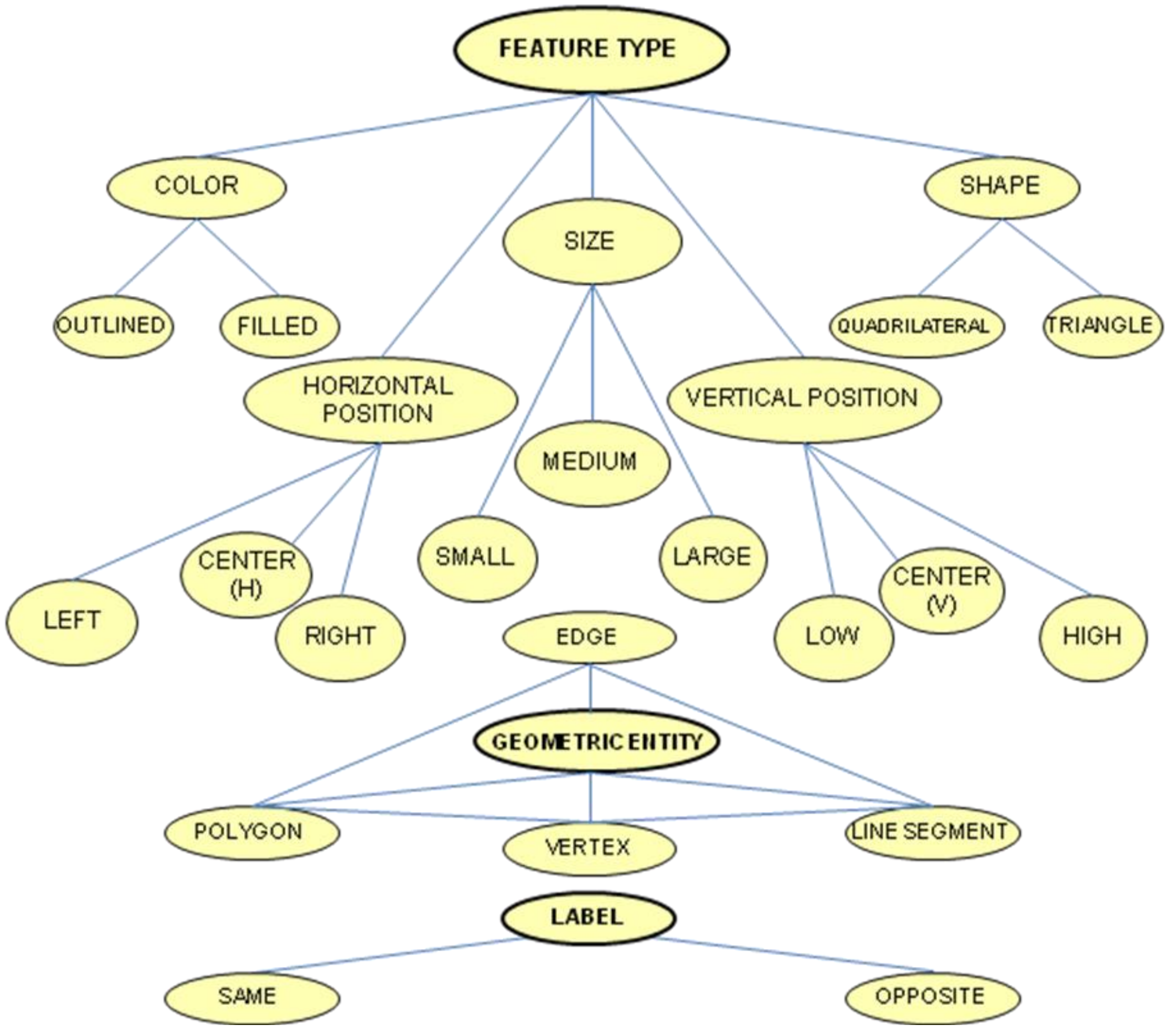

Figure 4. The concept network. Nodes with thicker border indicate concept categories. Fully active nodes can spread activation to their neighboring nodes through the links.

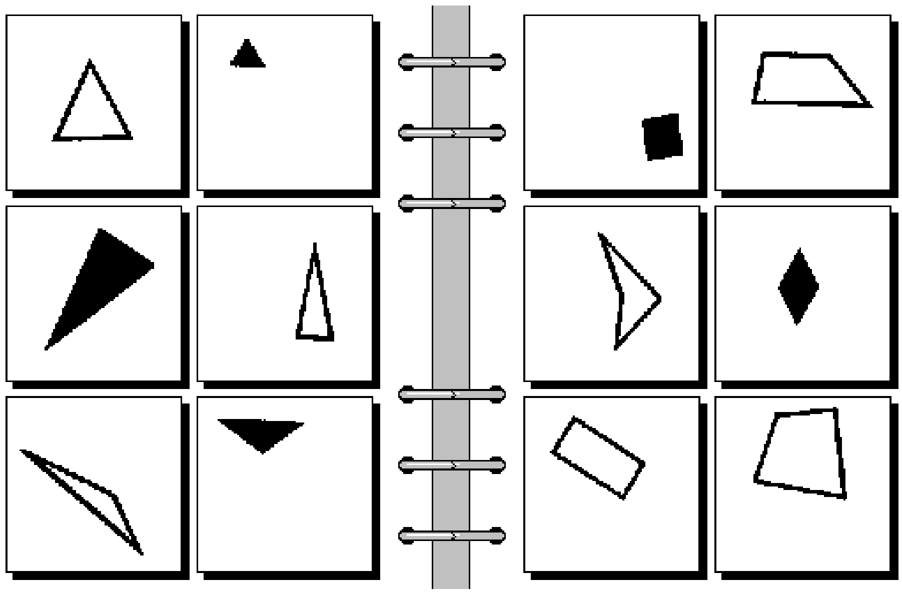

Figure 5. Problem BP#6 from Bongard's original collection [9] (images from H. Foundalis' web site [10]).

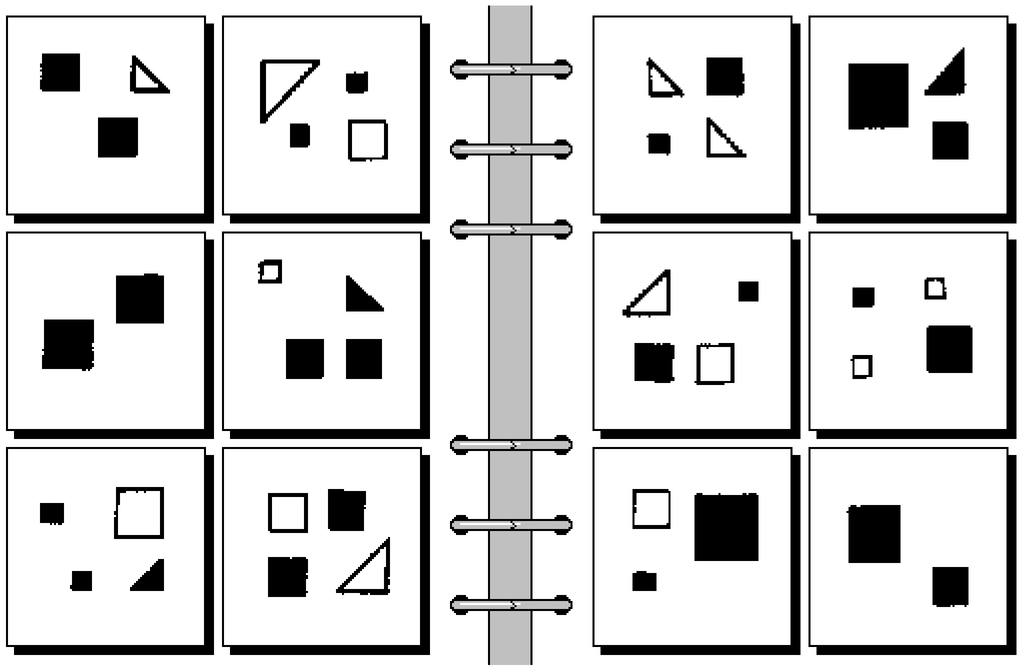

Figure 6. BP#58 from Bongard's original collection [9] (images from H. Foundalis' web site [10]).

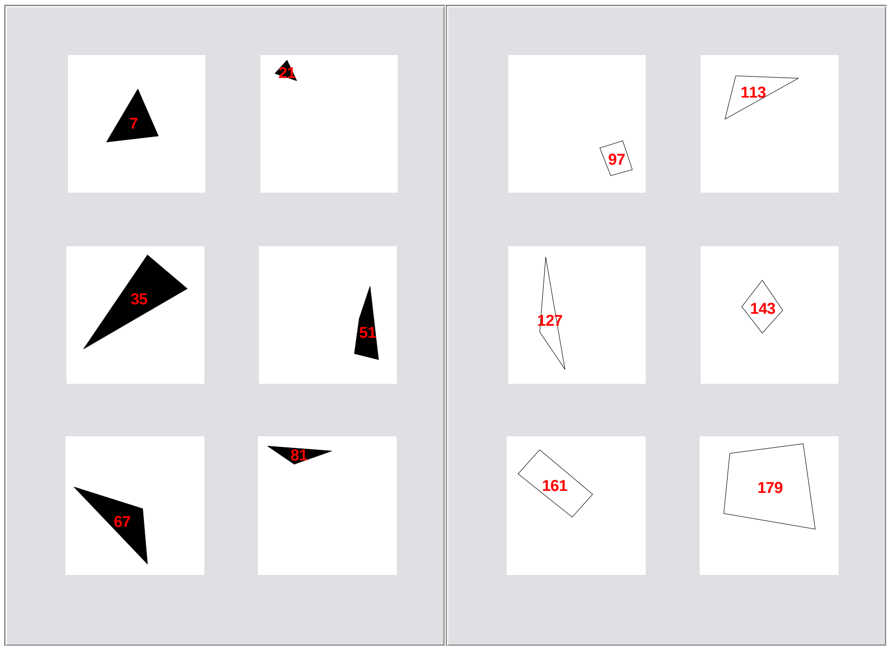

Figure 7. BP3: Filled vs. outlined.

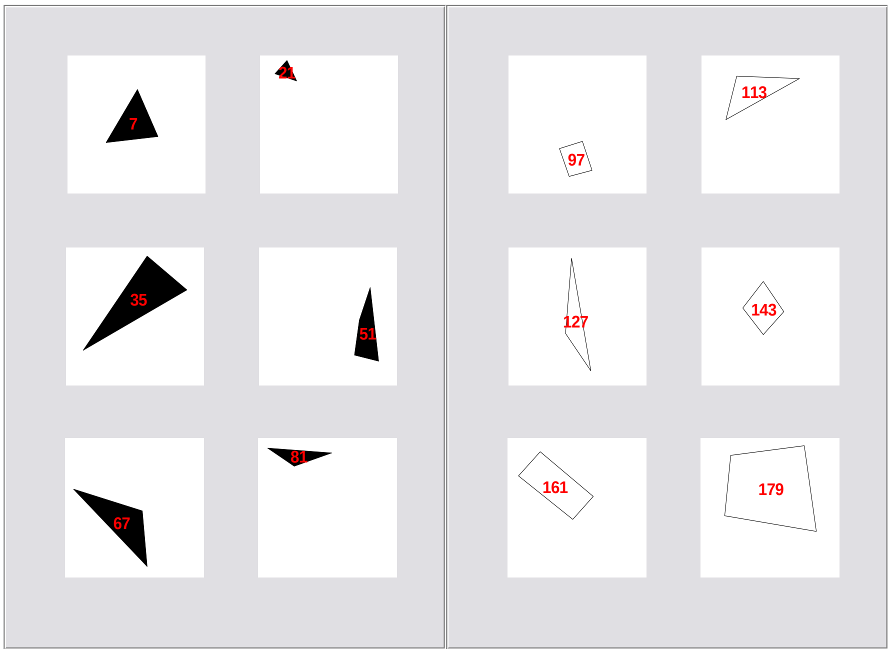

Figure 8. BP3.1: Filled vs. outlined. In this case, all the objects in the right set are also positioned in the middle of the horizontal axis.

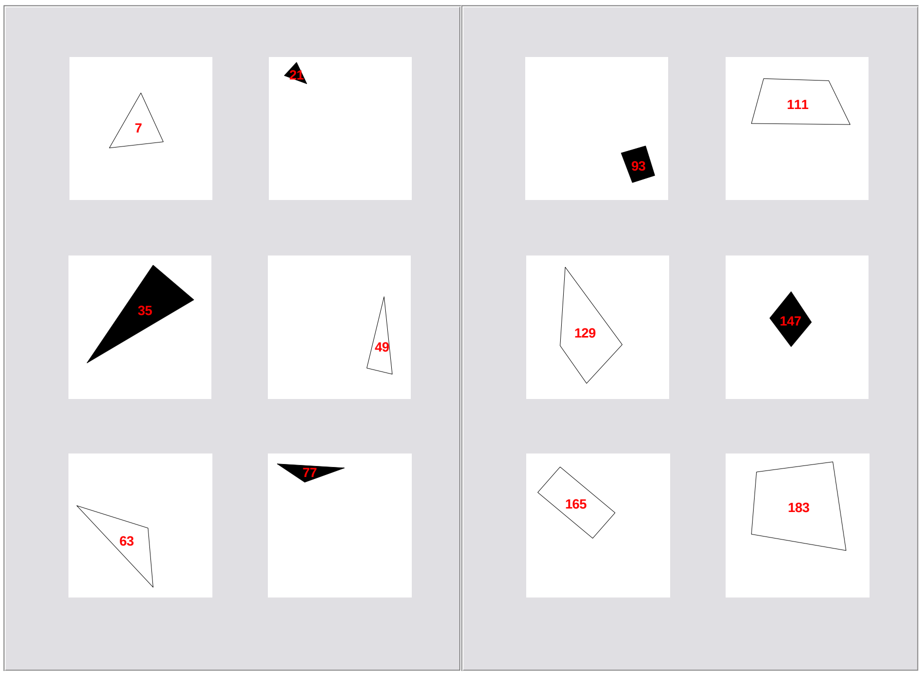

Figure 9. BP6: Triangle vs. quadrilateral.

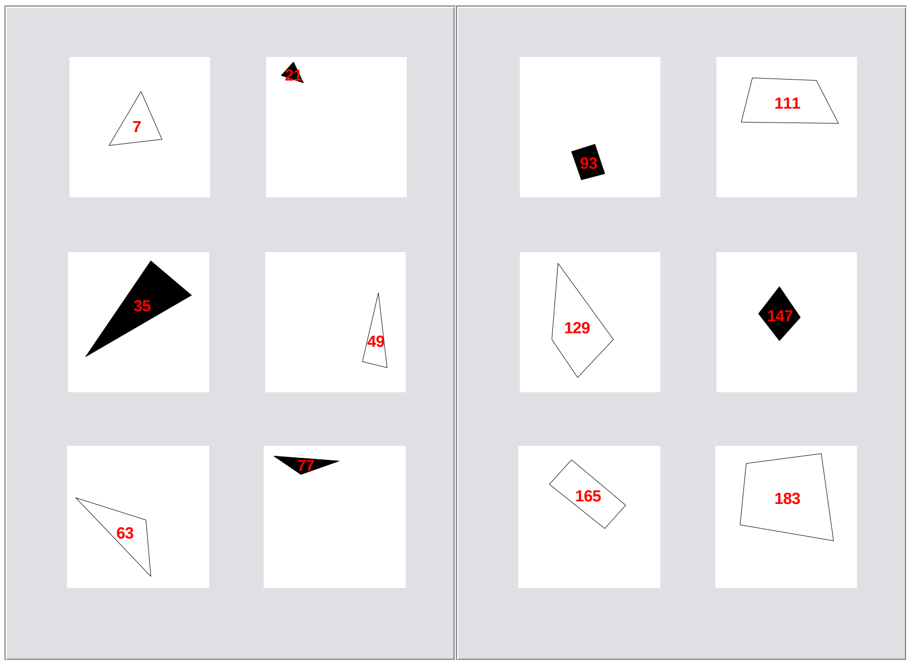

Figure 10. BP6.1: Triangle vs. quadrilateral. In this case, all the objects in the right set are also positioned in the middle of the horizontal axis.

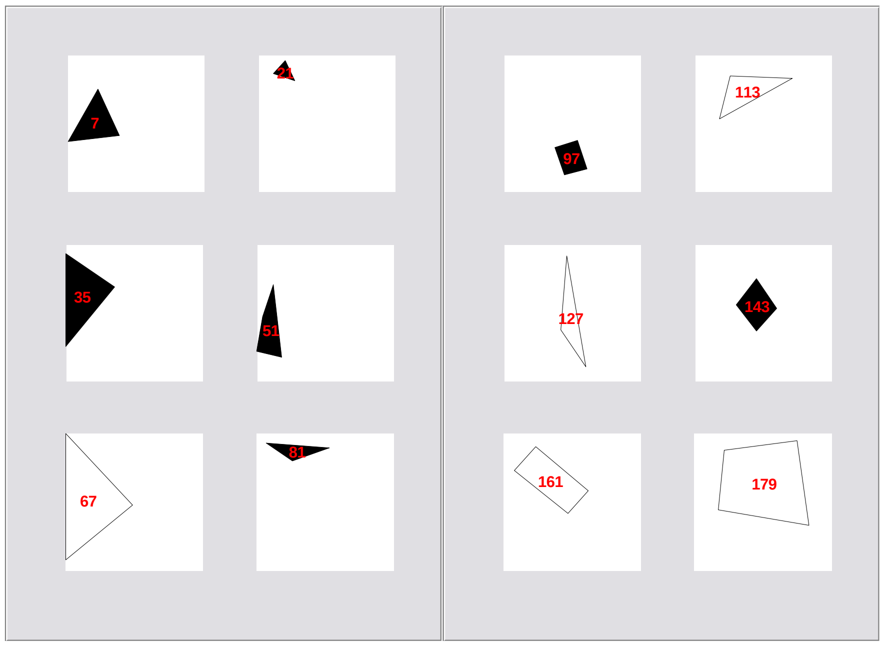

Figure 11. BPH: Left vs. horizontal center.

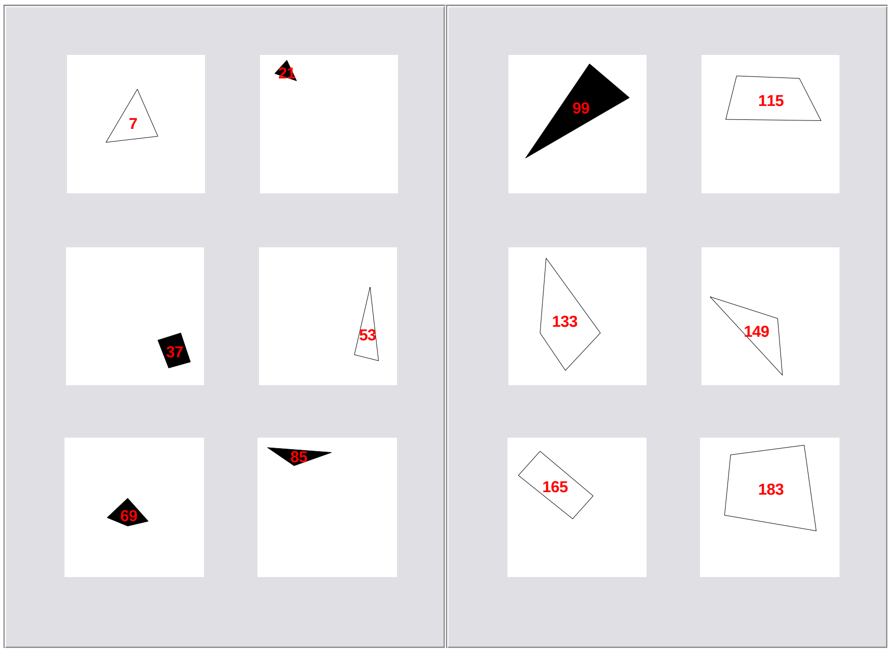

Figure 12. BPS: Small vs. large.

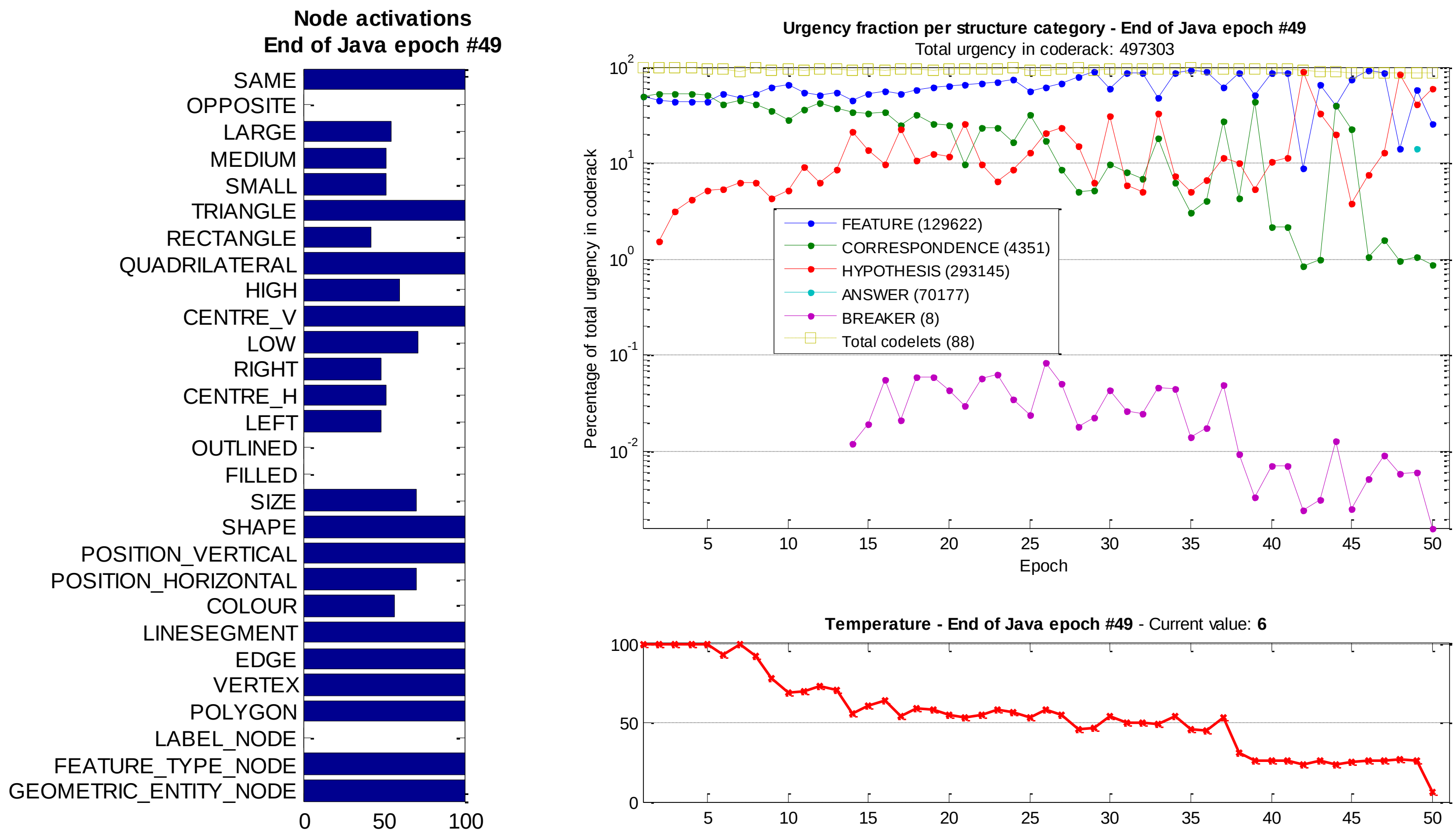

Figure 13. Node activations (left), time history of the urgency of codelets grouped by category structure (top right) and of the temperature (bottom right) at the end of one of the baseline runs for problem BP6.

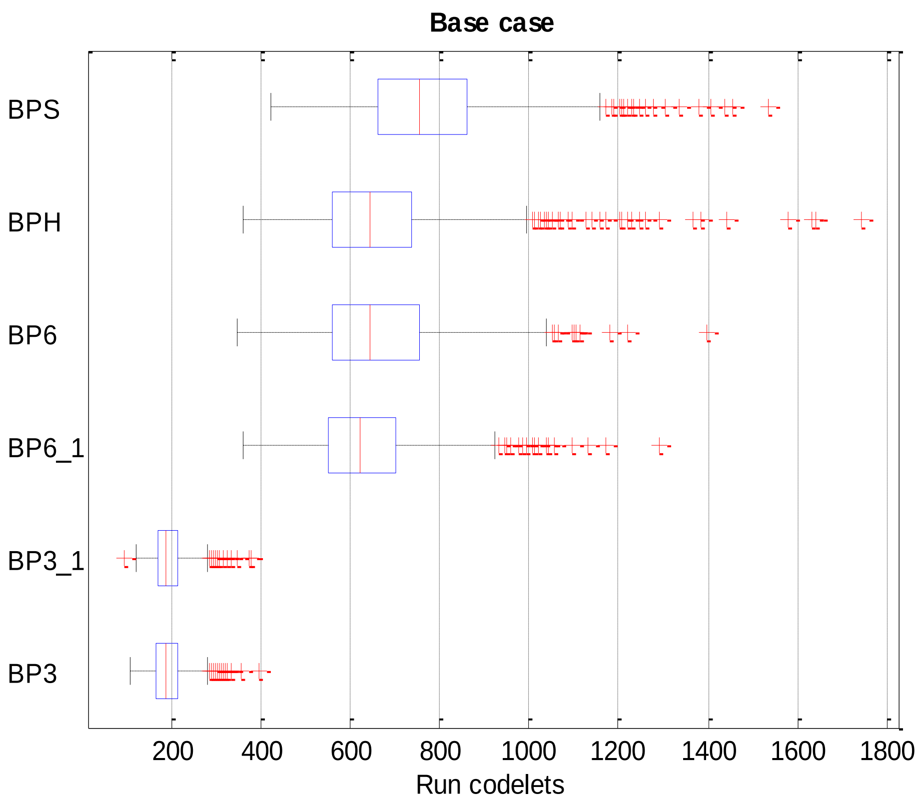

Figure 14. Boxplot of the distribution of the number of codelets run for each of the 6 baseline test problems.

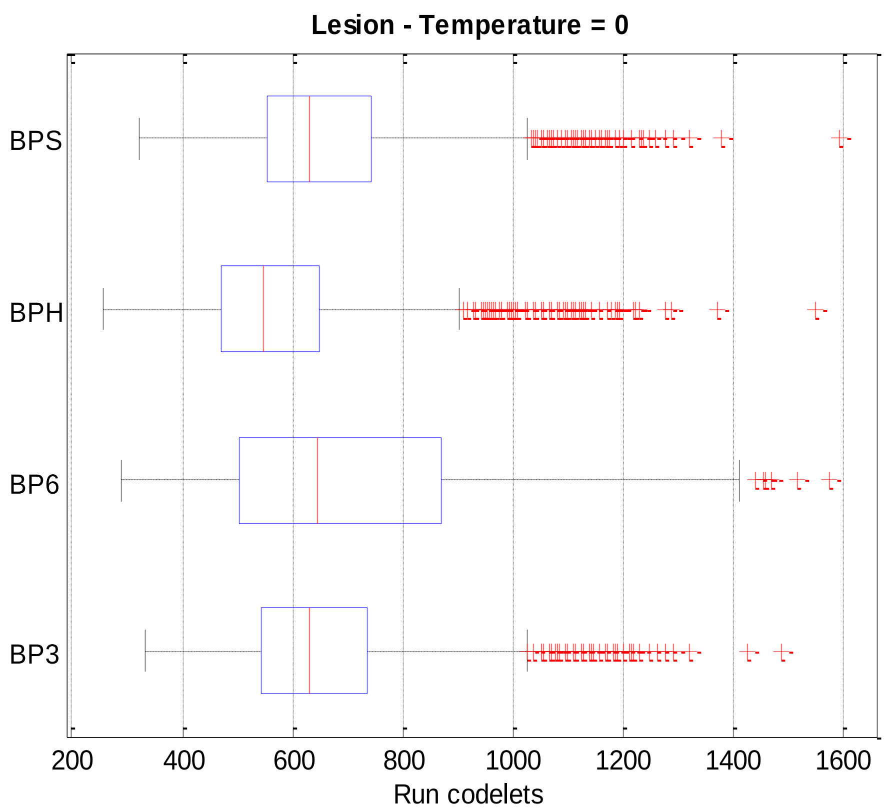

Figure 15. Lesion experiment: Temperature clamped at 0. Boxplot of the distribution of the number of codelets run for the 4 basic test problems.

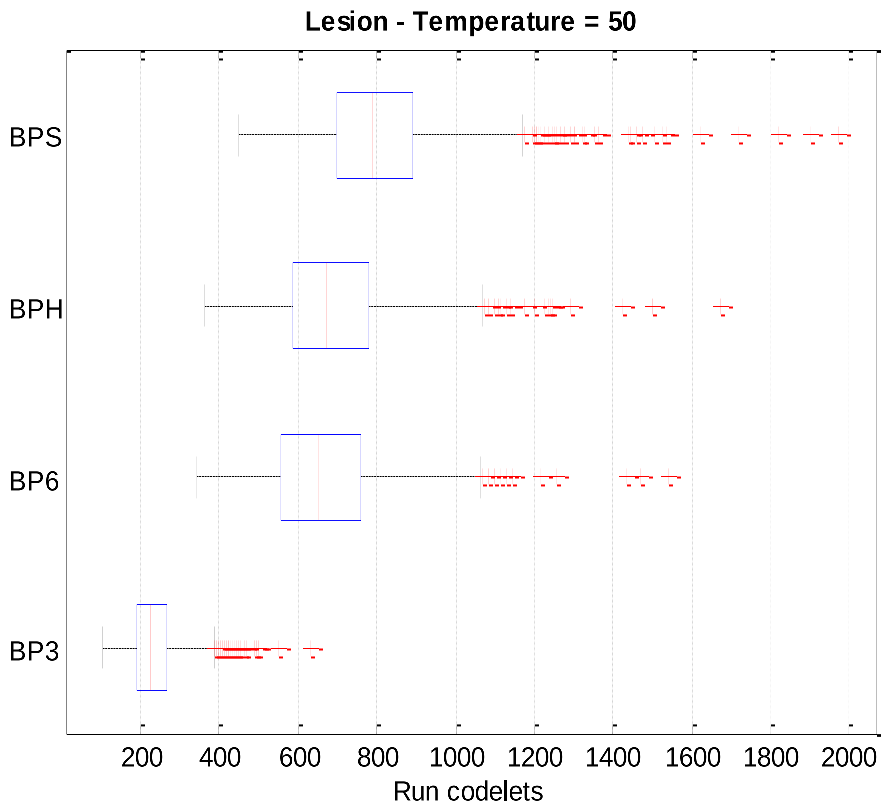

Figure 16. Lesion experiment: Temperature clamped at 50. Boxplot of the distribution of the number of codelets run for the 4 basic test problems.

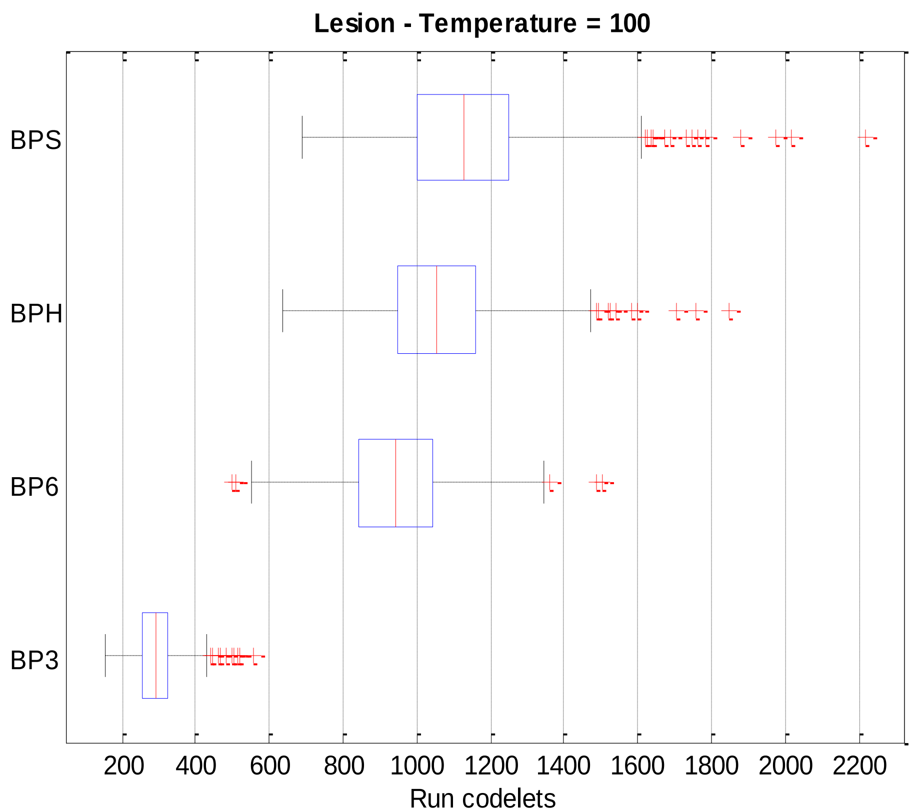

Figure 17. Lesion experiment: Temperature clamped at 100. Boxplot of the distribution of the number of codelets run for the 4 basic test problems.

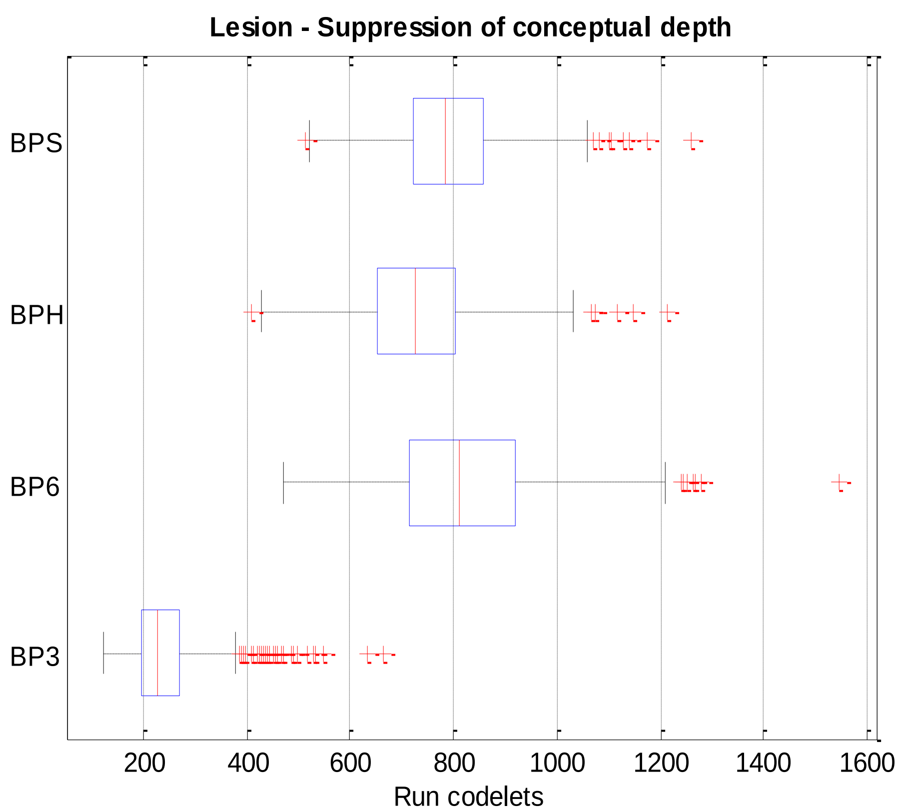

Figure 18. Lesion experiment: Suppression of differences in conceptual depths. Boxplot of the distribution of the number of codelets run for the 4 basic test problems.

Problem ID	$\bar{X}$	s
BP3 (Color)	193.89	36.70
BP6 (Shape)	665.93	143.00
BPH (Horizontal position)	669.08	166.34
BPS (Size)	777.76	161.89
BP3.1 (Color)	195.19	36.73
BP6.1 (Shape)	635.89	122.00

Table 1. Summary of the results of the baseline experiments. $\bar{X}$ is the sample mean, s is the standard deviation.

Problem ID		Baseline	T = 0	T = 50	T = 100
BP3	$\bar{X}$	193.89	675.24	235.80	292.38
	s	36.70	192.94	68.19	57.20
BP6	$\bar{X}$	665.93	706.32	670.17	944.35
	s	143.00	261.46	155.85	146.20
BPH	$\bar{X}$	669.08	592.00	694.57	1060.49
	s	166.34	186.09	157.97	162.99
BPS	$\bar{X}$	777.76	684.56	812.45	1137.39
	s	161.89	196.58	178.78	188.46

Table 2. Comparison of the mean ($\bar{X}$) and standard deviation (s) of the number of codelets run in each of the temperature-lesion experiments and in the baseline experiments.

Problem ID		Baseline	Conc. Depth lesion
BP3	$\overline{X}$	193.89	239.54
	s	36.70	66.77
BP6	$\overline{X}$	665.93	821.66
	s	143.00	151.43
BPH	$\overline{X}$	669.08	732.25
	s	166.34	113.95
BPS	$\overline{X}$	777.76	793.08
	s	161.89	104.17

Table 3. Comparison of the mean ($\overline{X}$) and standard deviation (s) of the number of codelets run in each of the conceptual-depth-suppression lesion experiments and in the baseline experiments.