ARC-AGI Final Report
CS7637

Joey Bishop
jbishop47@gatech.edu

1. FUNCTIONALITY AND DESIGN

The agent takes a heuristic approach to solving problems that involves parsing, testing, and solving. When given a problem, the agent parses the problem to derive relationships, with information such as whether a problem's training inputs and outputs share the same colors or have the same dimensions. Some relationships are problem-specific, and examples of these relationships are provided in section 2.1. The agent uses these relationships as prerequisites to applying its known problem types; if a problem satisfies the prerequisites, then it will attempt to apply the problem solution.

Solving a problem begins with the testing phase, where the agent attempts to apply the solution to each training set, deriving any information necessary to apply the test set. If the agent can sufficiently apply the solution to each training set, then the problem solution is considered valid, and the solution is applied to the test set. If no solution can be found in the agent's list of heuristics, then the problem's test input is returned as an output. As a result, there is a very high likelihood that an agent will not solve a problem for which there is no matching heuristic.

The agent does not store any data from problem to problem, and is not given any memory at initialization, aside from the heuristics and algorithms necessary to solve various problem types.

2. PERFORMANCE

Across the entire problem scope, the agent can solve 86 of the 96 problems. For Milestone B, the agent can solve 16 of the 16 training problems and 13 of the 16 test problems. For Milestone C, the agent can solve 16 of the 16 training problems and 15 of the 16 test problems. For Milestone D, the agent can solve 15 of the 16 training problems and 11 of the 16 test problems.

2.1. Agent Success

While the agent can solve all but one problem from the training set, it does not have exactly one heuristic for each problem. Some problems are variations of each other, and other problems build on shared logic. Examples of these problems, and how the agent solves them, are explored in the following sections.

2.1.1. Intersection

The Intersection problem type actually represents a family of problems with small variations, and the agent can reliably solve each of these. In simple terms, an intersection requires the agent to take N shapes, which may be separated by a solid line, overlay the shapes, and fill the shared space with the appropriate color. This sounds straightforward, but there are several alternatives to this problem.

For example, the shapes might be vertically or horizontally centered (Figure 1). Another variant lies in how to color an intersection: In some problems, the intersection is filled in black, with the remainder in green, but other problems require the inverse. Additionally, there might or might not be a solid line separating each shape. Further, the intersection might be a logical OR, AND, or XOR between the N shapes.

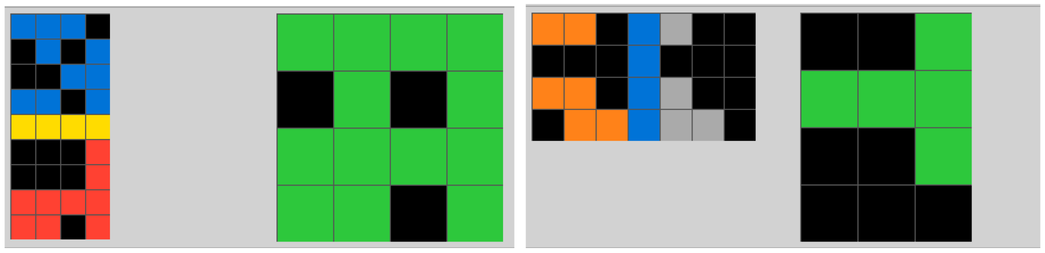

Figure 1*—A vertical intersection problem (left) that requires the solution to highlight the intersection in black, while another intersection problem (right) is horizontal and requires the solution to highlight the intersection in green.*

The agent solves this problem by first checking its prerequisites. If a problem's input shape matches the problem's output shape on one dimension, and the other dimension is N times that of the output dimension plus N, then the problem matches on size. In addition, the input must contain exactly three or four colors, and the output must contain two colors. If these requirements pass, then the problem is considered an intersection problem, and the dimensions are stored for the test phase.

To test, the agent takes the two shapes from each training input, ignoring the separator (if any), and overlays them. The overlay has four different strategies, involving what to match on and whether to invert the colors, and compares each with the training output, remembering which strategy was correct. If each training set can be solved with the same overlay strategy, then that strategy is applied to the test input.

2.1.2. Blob Highlight

Several problems feature a "blob"; a 2-dimensional closed shape (Figure 3). To identify blobs, the agent is equipped with a breadth-first-search approach applied to a given input or output. Once a blob has been identified, a "fill" is performed within the blob to determine if it is fully closed. This function is shared between several heuristics and algorithms.

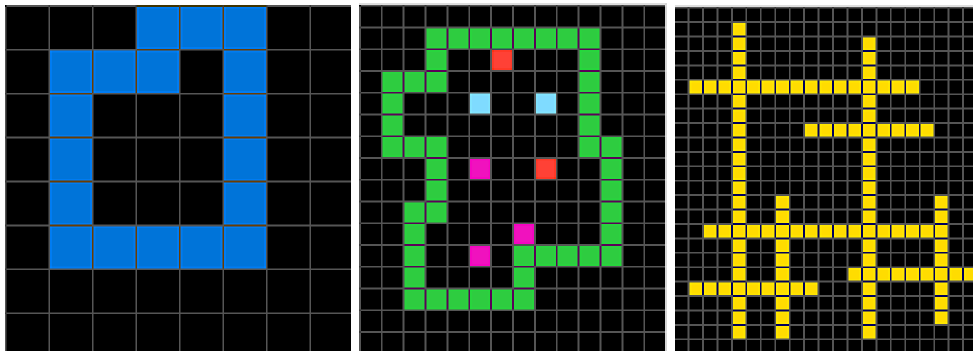

Figure 3*—Blobs can come in all shapes, sizes, and colors, but must be a closed 2D shape. They might be fairly simple, empty polygons (left), contain sparse colors within them (middle), or even intersect with itself (right).*

To match the Blob Highlight heuristic (Figure 4), the input and output for each training set must have the same dimensions and must *not* have the same colors. In fact, for each training set, the input can only contain a background color and a blob color, and the output must contain the same background and blob colors. The output may also contain two additional colors, which will serve as the exterior highlight and interior highlight colors. From there, blobs are compared between the training input and output. If the same closed blob exists in both the input and output *and* a highlight is applied to the output, then the training set passes. Highlight colors are stored to be used in the solve phase.

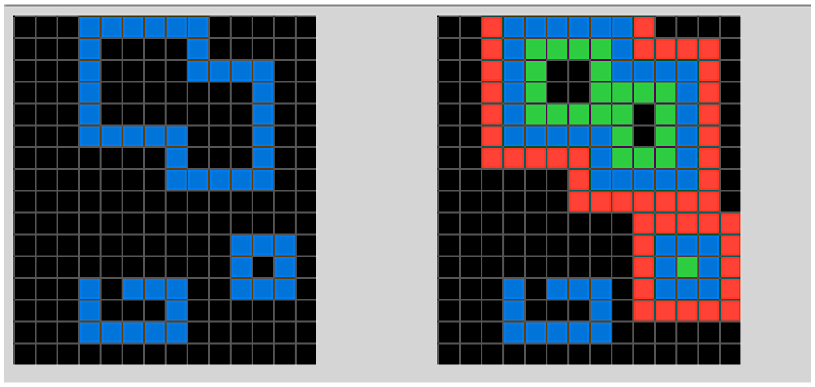

Figure 4*—The input (left) for a bob highlight problem features some fully closed blobs, and an open blob. To output (right) reveals the solution by providing the expected exterior and interior highlight colors.*

To solve the problem, the agent gets all blobs that exist in the problem test input, provided by the agent's blob identification algorithm. Then, the agent simply applies the appropriate exterior and interior highlight around the blob.

2.1.3. Wires

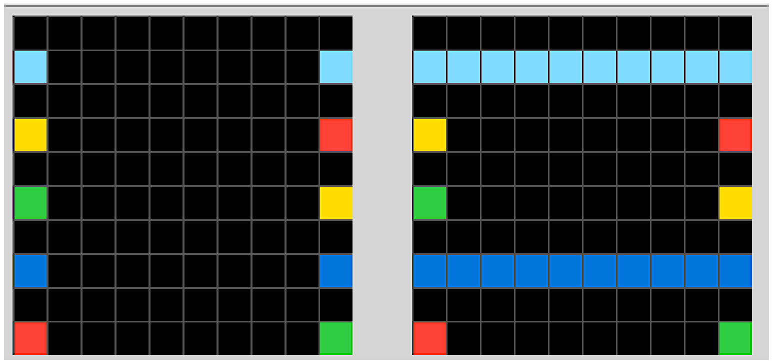

Figure 5*—In a Wires problem, only rows where the left and right edges share the same color are connected. In the first row, cyan exists on both edges in the input (left), so a horizontal wire is drawn between them in the output (right). The second row contains yellow and red, so no line is drawn.*

In the Wires problem type, the left and right edges of the input feature rows of alternating pixel colors, and the output contains horizontal "wire" lines that connect two pixels of the same color on the same row, as seen in Figure 5. These problems require the input and output of each training set to share the same colors and dimensions, the input's left column matches the output's left column, the input's right column matches the output's right column, and the center of the input is a background color.

To test, the agent simply finds rows where the same color, aside from the background color, are present, and fills the entire row with that color. If the agent can solve each training problem with this method, then the same algorithm is applied to the test input.

2.2. Agent Failures

The heuristic approach for this agent has enabled it to solve all but one problem from the training set. However, it is clear that it does not appropriately abstract all problems to recognize certain variations. The failing training problem and an unsolved test problem are detailed in the following sections.

2.2.1. Blob fill with most common color

The agent can recognize this algorithmically complex problem type, but is unable to reliably provide a solution. The heuristic for this problem relies on the existence of blobs and color relationships between input and output. Specifically, blobs must exist between both input and output, and the output cannot contain more colors than the input. Fill colors in the output are also compared with pixel colors in the input.

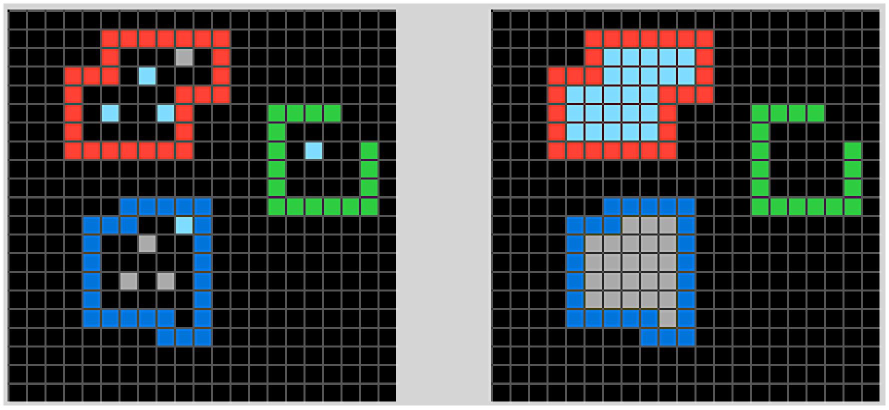

Figure 6*—In the "Blob fill with most common color" problem, a set of blobs and open shapes are given (left) with some floating pixels. In the solution (right), each blob is filled with the most common enclosed pixel color. All other pixels are removed.*

To solve this problem (Figure 6), the agent must execute several steps. Firstly, the agent must recognize all blobs in the test input. Then, the agent must retrieve the most common pixel color enclosed by each blob. Following, each blob must be completely filled with its corresponding pixel color. Finally, all remaining pixels must be cleared from the grid.

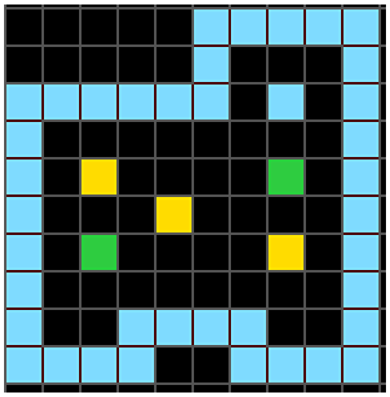

Figure 7*—A blob where the bottom surface is concave.*

The algorithm fails to properly fill complex blobs, such as those with concave surfaces, as in Figure 7. This introduces additional complexity in filling a blob that results in errors. In addition, recognizing stray pixels in the grid proved to be more complex and computationally intensive than expected. As a result, the agent is unable to reliably solve this problem when there are complex blobs or stray pixels.

2.2.2. Spiral

In the Spiral problem type, only an empty input is provided, and a color spiral filling the space of the grid results in a proper solution, as seen in Figure 8. The agent's heuristic for this problem type is flexible enough to recognize any color for the spiral color, but its rigid in that it expects the spiral to always begin in the top-left corner and move clockwise. This rigidity in the heuristic prevents the agent from even applying the algorithm to Spiral problems where the spiral might begin in a different corner or the spiral moves counter-clockwise.

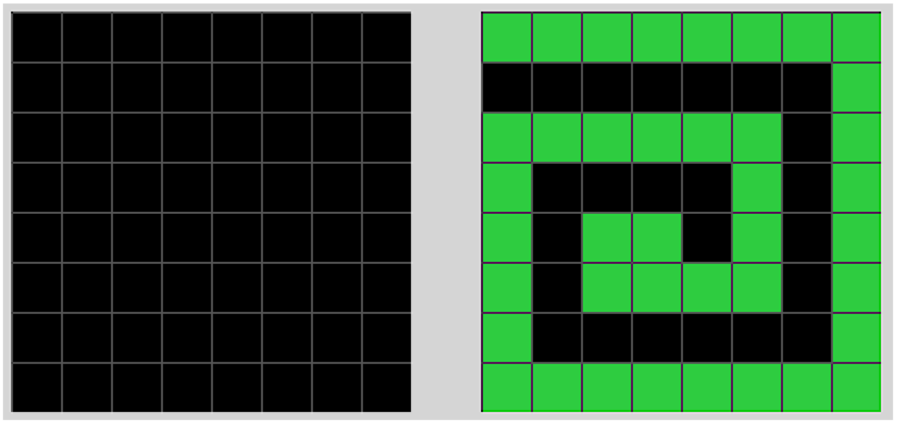

Figure 8—A Spiral problem type begins with an empty grid (left) and finishes with a spiral filling the whole space (right).

3. HEURISTIC APPROACH

The overall agent design is a heuristic approach. Each problem type is recognized with a set of criteria, or heuristics, that limit the agent from applying a certain solution type unless that criteria is met by the problem. The heuristics are abstracted, allowing for variants in dimension, color, shape counts, and more between examples of problem types. As a result, the heuristic really tests *relationships* between training sets and training set inputs and outputs. For example, two training sets might have different colors, but if each training set's input has one less color than its output, then the training sets might belong to a certain problem type.

This approach has grown over the course of the project, but has remained to its core since Milestone B. Similar heuristics have been grouped into certain classes in the code, which enables all of those heuristics to leverage shared logic or properties. For example, if two different heuristics need to identify blobs, then they will be grouped into a blob class and can each rely on shared logic for identifying blobs.

Each heuristic has an accompanying algorithm for solving the problem. Algorithms range in complexity from rotating an input to scaling, transforming, counting, and recoloring shapes in the input. Similar to shared heuristic class logic, shared functions are available to algorithms that must perform similar mutations or calculations.

4. AGENT VERSUS HUMAN

At a high level, the agent takes a similar approach to solving known problems as a human would. Both a human and the agent perform some preprocessing over the problem, such as discerning relationships in dimensions, colors, and shapes between inputs and outputs, to derive meaning. Both the human and the agent can select an algorithm to solve the problem based on the relationships and meaning derived. The nature of the algorithm might be different between agent and human, as we do not normally think in terms of color masks and matrix multiplication, but the process of selecting an algorithm is similar.

Unfortunately, the similarity between human and agent is broken when a problem solution is unknown. Because the agent behaves heuristically and is limited to its given algorithms, it does not have the capability of creating new problem solutions. A human, on the other hand, can often create and apply a new algorithm to solve a novel problem. From there, the novel problem type and algorithm are stored in memory for later use. The agent does not have any capabilities for this and is unable to solve the problem.